COMPLEXITY AS A FACTOR OF QUALITY AND COST IN LARGE SCALE SOFTWARE DEVELOPMENT

Joe Newton Harris

NAVAL POSTGRADUATE SCHOOL
Monterey, California

THESIS

COMPLEXITY AS A FACTOR OF QUALITY AND COST
IN
LARGE SCALE SOFTWARE DEVELOPMENT

by

Joe Newton Harris

December 1979

Thesis Advisor: N. F. Schneidewind

Approved for public release; distribution unlimited.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Complexity as a Factor of Quality and Cost in Large Scale Software Development		5. TYPE OF REPORT & PERIOD COVERED Master's Thesis; December 1979 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Joe Newton Harris		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Naval Postgraduate School Monterey, California 93940		12. REPORT DATE December 1979 13. NUMBER OF PAGES 97
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) Naval Postgraduate School Monterey, California 93940		15. SECURITY CLASS. (of this report) Unclassified 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Software Complexity, Software Quality, Software Cost Estimation

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The impact of complexity on software quality and costs is examined. Historic and current issues relating to complexity in the software development and software cost estimation processes are reviewed. Select complexity models and metrics are described and briefly analyzed. Finally, an argument is presented in support

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-014-6601 |

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Date Entered)

of McCabe's Directed Graph Model as a useful software management tool in controlling complexity, formulating a test strategy and allocating resources.

Approved for public release; distribution unlimited.

Complexity as a Factor of Quality and Cost in Large Scale Software Development

by

Joe Newton Harris
Lieutenant Commander, United States Navy
B.A., Duke University, 1968
MBA, National University, 1975

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN MANAGEMENT

from the

NAVAL POSTGRADUATE SCHOOL

December 1979

ABSTRACT

The impact of complexity on software quality and costs is examined. Historic and current issues relating to complexity in the software development and software cost estimation processes are reviewed. Select complexity models and metrics are described and briefly analyzed. Finally, an argument is presented in support of McCabe's Directed Graph Model as a useful software management tool in controlling complexity, formulating a test strategy and allocating resources.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

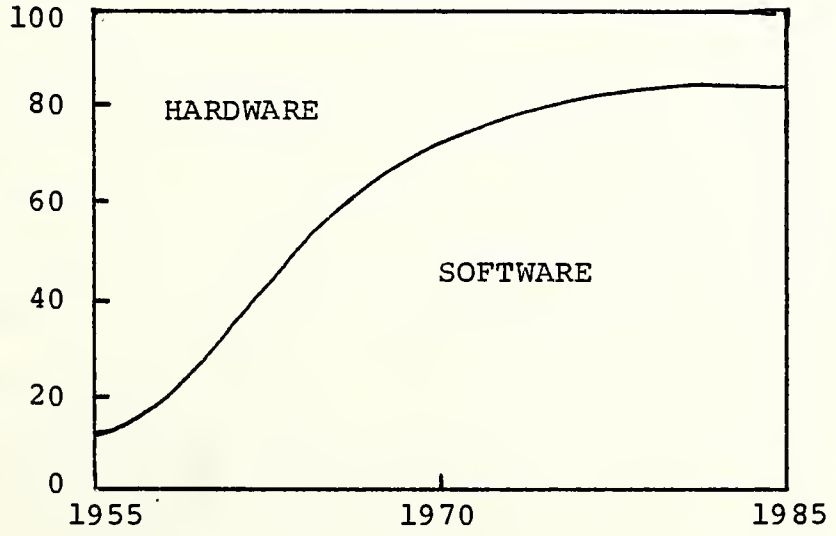
ACKNOWLEDGEMENT

I wish to express special gratitude to Professor Norman F. Schneidewind for his continuing guidance and criticism during the research. Also of great assistance despite busy personal schedules were Mr. Ray Wolverton (TRW), Mr. Ben Ogle (Hughes Aircraft Company), Mr. Robert McGhie (Hughes Aircraft Company), Mr. Dwayne Smith (FCDSSA), and Mr. Nick Bayerle (FCDSSA). Finally, the assistance and support of my wife, Michael Ann Harris, was indispensable.

I. INTRODUCTION

Total procurement costs of large scale computer systems are conventionally divided into costs associated with production of hardware components (computers and peripheral equipment) and costs associated with software development (program design, coding, test, maintenance and documentation). While hardware costs dominated in early computer models, the combined effects of improved cost-reducing technology in the production of hardware and marked rises in the costs of labor to develop programs have resulted in a dramatic reversal in this situation today. [1, 2] Indeed, if these trends continue, software costs will converge to approximately 90% of total computer procurement costs in the mid 1980's (see Figure 1). [3]

FIGURE 1 HARDWARE/SOFTWARE COST TRENDS (3)



From a management perspective, the steady incremental reduction in hardware unit costs has been a concrete manifestation of technological gains and increased productivity. Although management problems have arisen, they have been the livable difficulties of a high growth industry meeting each new challenge with multiple technical solutions, thereby constraining managers only by their abilities to adapt to and harness new opportunities.

Conversely, the rising costs of software are directly indicative of a critical mismatch between complex needs and limited technical abilities. Trends and pressures leading to this situation have existed from the beginning attempts to apply general purpose computers to progressively more complex and larger problems of society. As awareness of a large number of system development failures and near failures increased in the recent past (illustrated by cost overruns, schedule slippages and performance degradations), a growing appreciation of the scope of unsolved technical and productivity problems began to emerge.

This awareness was well summarized at the 1973 "Symposium on the High Cost of Software" in a statement that continues to apply today: "Progress in software technology has been very slow, but demands for software production are increasing in volume and complexity. Such demands have clearly outstripped the technology, with very costly results. Production of new software products suffers great overruns in cost and delivery time, and quality is often deficient in

correctness, modifiability and transferability. The maintenance costs of old software products may be an order of magnitude larger than production costs, due to poor original design and production." [4]

In order to close the gap between existing software technology and production demands, a number of noteworthy programming/management techniques have been developed and implemented with varying success. These developments include computer aided specification generation, top-down design, structured programming, chief programmer team, egoless programming and program walkthrus.

Also imbedded in the historic problems of developing large scale software has been an inability to produce accurate project cost and schedule estimates and a corresponding managerial failing to correctly assess risk and critically evaluate estimates and associated underlying assumptions presented by subordinate software estimating groups. The cumulative project costs of developing and maintaining large scale system software are determined by a myriad of interrelated variables including the quality and stability of original design specifications, the relative difficulty of the technical problems involved, the productivity of the programming group available, and the traditional project management skills of efficient resource direction and utilization. The relative distribution of available resources over production phases varies with each project. However, studies have indicated

the average % resource requirement distributions summarized in Table I. [2]

TABLE I

Percentage Distribution of Resource Utilization

DEVELOPMENT PHASE*	ANALYSIS & DESIGN	PROGRAM WRITING	TEST & INTEGRATION
PROJECT TYPE			
Military Command & Control System	35	15	50
Space Oriented System	35	20	45
IBM 360 Operating System	35	15	50

*NOTE: This table ignores maintenance expenses incurred after system deployment.

The occurrence of the proportionately high cost factor in the test and integration phase as indicated in this summary has come as an unpleasant surprise to many project managers and to those supplying project funds. The chronic underestimating of these costs is most directly attributable to a pervasive lack of appreciation for the extent of required managerial involvement and severity of potential pitfalls associated with the iterative process of software quality assurance. When a manager underestimates the dollar and time requirements of the test phase, he often exacerbates them by embarking on an inadequate initial effort which is essentially wasted. System quality must be a focal management concern throughout a project. Costs to recover during testing for earlier management control mistakes are normally prohibitive. Further,

indirect costs, which do not appear as part of the project, often result from pressured attempts to shortcut the test phase. Such costs include late deliveries and resulting slipped schedules, delivered system degradations and associated spiraling life-cycle support costs as well as difficulties in funding follow-on projects due to mistrust of presented estimates and fear of further overruns.

Recently, significant efforts (referenced later) to improve software management performance have centered on recognition of software complexity as a quality and cost determinant. If complexity can be measured, controlled (e.g., by threshold) and shown to reliably predict the probable effort required for error detection and correction, an important tool will be available in the effort to understand and manage large scale software development costs.

This thesis is aimed at investigating the impact of complexity on software quality and costs and the potential ability of management to exploit this impact. In conducting the investigation, the cornerstone work by McCabe in applying the cyclomatic number from directed graph theory as a measurement proxy for software structural complexity and the supportive experimental work at the Naval Postgraduate School supervised by Schneidewind were particularly useful. Further, field trips were made to three software production facilities (TRW, Redondo Beach, Ca.; Hughes Aircraft Co., Fullerton, Ca,; U. S. Navy's Fleet Combat Direction System Support Activity (FCDSSA), San Diego, Ca.). These field trips served to

determine current cost estimating and resource allocation procedures and to validate by interview the existing confidence levels in complexity or other cost predictors by those currently involved in this effort. Results of these trips are cited as appropriate. While user/customer issues are recognized where relevant, the perspective of the development agency is emphasized.

Chapter II discusses issues concerning the development and control of large scale software. Chapter III summarizes some select aspects of complexity and complexity metrics relative to software. Chapter IV reviews a recent experiment relevant to the application of complexity measurement theory to management practices. Chapter V describes the resource estimation problem and suggests a management approach to resource allocation utilizing the cyclomatic number metric as a guideline. Finally, Chapter VI offers a summary and conclusions.

II. SOFTWARE DEVELOPMENT AND CONTROL

A. NATURE OF SOFTWARE

Software includes both the conceptual solution to a proposed problem and the documentation required to translate this solution into a workable computer program. Its nature is marked by a lack of measurable physical characteristics. The management of software development historically suffered because essential similarities and differences between software development and traditional hardware design and production were not well understood. Management understanding of these comparisons is essential to controlling software quality. The most important of these similarities and differences are listed below: [5, 6]

- While hardware engineers utilize a sequence of development prototypes enroute to the production model, software projects often begin with a concept that the first version developed will be the delivered product. This concept is naturally reflected in personnel, monetary and calendar-time estimates and expectations. History has indicated a definite need for iteration in software development analogous to the hardware development model.

- The institutionalized sequence of hardware production provides natural control points for management review and design freezes. Software development has no such natural points and often suffers from changes throughout. Design

freezes are essential for ordered software development and must be arbitrarily imposed by management.

- Hardware engineers expect designs to be fully tested by well understood procedures and customarily prepare test plans. Although pressures to formally test software are now substantial, testing techniques are still at an innovative stage and much quality evaluation remains highly dependent upon individual programmers.

- Hardware is essentially composed of standard parts with stable performance characteristics. Software sub-routines are often new, innovative and not fully understood.

- Hardware reliability is related to the passage of time much differently than software reliability. With hardware, "An accumulation of stresses is reached which causes a component to fail." [6] Conversely, a software error exists due to programmer activity or inadequate specification. "The amount of time (labor and machine) involved in error detection and the probability of error detection are a function of test time, type of test, and choice of test data." [6] Barring major modifications, software boasts an indefinite life, continuing to improve (decreasing error rate) with mounting testing and use.

- A software module with a detected error cannot be pulled off-line, replaced with a working unit and repaired. It must be repaired in order to fix the system. (The idea of fault tolerant programming incorporating redundant modules has been used in real time applications requiring high-reliability.)

- Correction of a software fault generally results in a new software configuration.

- In the process of making additional copies of software, no imperfections or variations are introduced (save for a class of easily checked copying errors).

B. SOFTWARE QUALITY

The quality of software has many aspects. Each aspect can become overriding in importance, depending upon the program application and the user's intention. During development or design change implementation, ease of revising (and verifying) is important. During deployment, ease of operating is paramount. Similarly, if a need develops to adapt the software to another system (hardware, software or both), ease of transition will be an important attribute. Table II [7] lists 11 software quality factors within this framework.

Although Table II does not necessarily provide a complete list of quality factors, most additional terms or criteria of software quality can be related to those described.

TABLE II

Software Quality Factors

QUALITY CATEGORY	QUALITY FACTOR	- DEFINITION
I REVISION	(1) Maintainability	- Ease of locating & correcting errors
	(2) Flexibility	- Ease of modifying program
	(3) Testability	- Ease of adequately testing (includes traceability: ease of linking requirements to design and code)
II OPERATION	(4) Correctness	- Extent to which user requirements are met
	(5) Reliability	- Extent of accurate and consistent operation
	(6) Efficiency	- Relative optimal use of computing resources and code
	(7) Integrity	- Relative ability to control unauthorized data access
	(8) Usability	- Ease of learning, operating, preparing input and interpreting output
III TRANSITION	(9) Portability	- Ease of transfer from one hardware configuration or system software environment to another
	(10) Reusability	- Ease of applying to other programs (relative to packaging and scope)
	(11) Interoperability	- Ease of interfacing with another system(s).

The dominant aspect in all software quality factors is program complexity. In general, as program structures become more complex, the probability increases of encountering difficulty in revision, operation and transition. [1, 2, 8] Accordingly, controlling complexity is a key concern of management in project development.

C. DEVELOPMENT CYCLE

In order to accurately estimate and/or effectively control a large scale software project, the development cycle must be understood. Although different authors and managers vary in some detail or nomenclature, the industry's successes and failures have distilled a generally accepted progression of activities necessary to produce a large scale computer program. The major phases of interest are comprised of the following:

- Analysis
  - planning
  - requirements definition
  - specification
- Design
- Coding
- Integration and Testing
- Life Cycle Support/Maintenance

Figure 2 (substantially from [5]) depicts this development cycle in chronological detail. It is important to note the iterative nature of this cycle, represented in Figure 2 by connecting arrows. Often events in one phase, such as testing, stimulate reworking of problems in a previous phase, such as coding or even design. Additionally, it is common for significant phase overlaps to occur at certain stages (e. g.

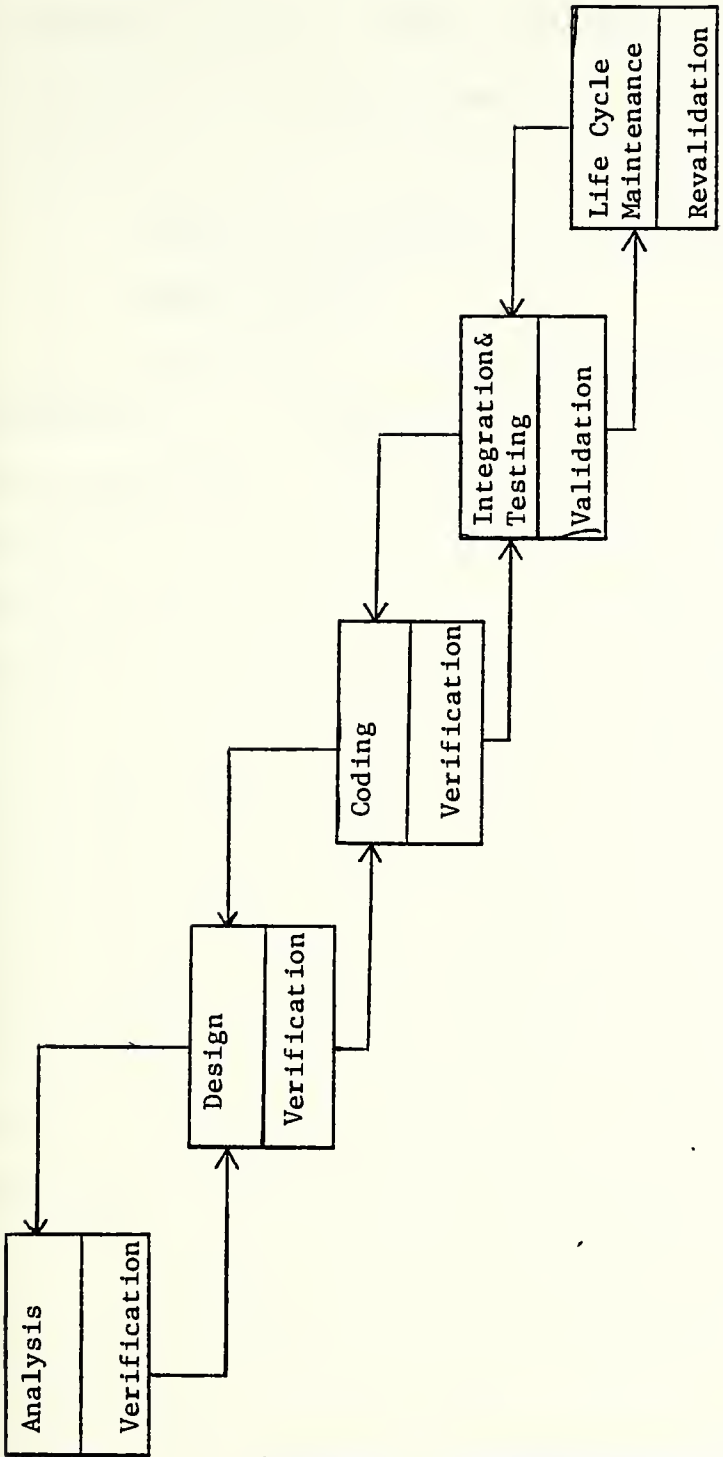


FIGURE 2. SOFTWARE LIFE CYCLE

conducting testing prior to completion of all coding). As mentioned earlier, freezing design at some point is an essential function of management if the project is to be completed on schedule, and at planned-for cost.

1. Analysis (planning, requirements definition, specification)

The initial or analysis phase of a project can be extended over a considerable time period and include several key activities. The user/customer generates requirements during this stage and communicates them to (potential) project engineers. This is an iterative process and is frequently stimulated by engineers (or marketeers) describing what is possible to users. Two issues, validity of user need and feasibility of solution, must be resolved prior to promulgation of user requirements. The organization's standard cost-effectiveness justification process is necessary for the first, while an independent feasibility study is normally initiated to settle the second. When either process is circumvented, continuity of future organizational decisions and actions is jeopardized.

When user requirements are articulated, they become inputs for resource utilization estimates which, along with resource availability issues, form the major considerations of development agency top management review. This review determines if the organization will pursue involvement (e.g. respond to Request for Proposal) and must assure that high

risk projects are discarded. [9, 10] Appropriate assessment of potential system/program complexity is crucial to the accuracy of this risk determination.

If the decision to continue is reached, specification (i.e., translating requirements to guidelines for development) is commenced as a final activity in the analysis phase. A management review concluding the analysis phase avails development agency management a final opportunity to determine project continuance/termination prior to major resource expenditures. Documentary output of the specification effort will support this review and guide future progress of the project. It is composed of detailed administrative and technical documents which are meant to form the bases of all user-developer contracts. [9]

2. Design

The design phase covers all remaining efforts required to complete the technical solution in light of specifications and imposed constraints. It culminates in describing the best technical solution in terms that will facilitate coding. [11] Short stopping errant designs is essential to avoid massive, costly rework in a project's latter stages. Customer/user involvement in the evaluation process is mandatory to ensure continuing communication and to engender commitment to approved designs before they key follow-on effort.

3. Coding

Coding includes both the translation of designs to computer language and the process of documenting developed programs.

The coding and design phases are particularly interrelated and can be interspersed with a technical review of each subroutine to track and assure progress. An important management review is often held at the end of coding with summary data available from all preceding technical reviews. As a rule, a large percentage of planned project development funds have been expended at this stage. With such a commitment from the customer, project termination is rare once coding is complete. Thus, while earlier reviews concentrate on project continuance, the major objective now is "to maintain schedules and budget by" shifting manpower from less important activities to critical tasks, canceling or delaying features, allowing standard practices to short-cut, and if all fails, to immediately publish a schedule or budget "increase." While these are management concerns throughout the project, they become particularly germane at the completion of coding when a genuine, albeit tenuous, attempt is made to refine total resource requirement estimates. [9]

4. Integration and Test

This phase includes the processes of merging all system/software components and demonstrating performance quality.

Daly [9] identifies four stages of software testing as follows:

- Segment or unit testing verifies the operation of individual design functions as they are developed.
- Module testing assesses segments combined into modules.

- Integration testing evaluates the progressive activity of merging all software into a single program.
- Systems testing assures that the software and all associated hardware in the total product system can function satisfactorily together. During this process it is important to exercise each function under full load or stress conditions such that the environment to be experienced by the user is simulated as closely as possible.

Both unit and module testing may be included in the coding phase. Integration and system testing are often duplicated, first by the developing agency and then during acceptance tests by the user. A potential for time and money savings exists here by having the user present for final integration and systems tests. It is an important opportunity for the user to gain familiarity with the program and confidence in the developer and program quality. Further, such arrangement may result in satisfaction of select acceptance requirements and thus cut test time. As the danger of compounding existing disagreements is great, this opportunity should only be exploited if, in the judgement of management, undue strain will not be placed on the customer relationship.

Testing requirements must be written and agreed upon very early in the development cycle. It is imperative that they reflect user involvement and represent a thorough yet cost effective attempt to verify system performance. [6]

5. Life Cycle Maintenance

Once acceptance testing has been satisfactorily completed, and the system transferred to the user, the life cycle support or maintenance phase begins. As Figure 3 [5] indicates, this activity constitutes a growing majority of development costs. Update maintenance, initiated by changed specifications resulting from altered user requirements must generally be handled on demand. Unless such alterations can be anticipated through close involvement with user, little can be done to minimize these changes. However, corrective maintenance is a preventable evil. Improvement techniques in all other phases must be invoked to minimize the occurrence of operational "bugs." These errors are even more costly to correct than those discovered in testing for the following reasons [6]:

- Problems are often more complex.
- Problems are reported as system malfunctions by operators not knowledgeable of data required to duplicate failure--effort must be expended to translate problem symptoms into systems error. (Operator training may improve this problem.)
- Problems are usually addressed by maintenance programmers who are unfamiliar with program development and must spend excessive time reviewing detailed code (normally not top personnel [5]).
- Another round of problem definition, design, code, test and full documentation is initiated.

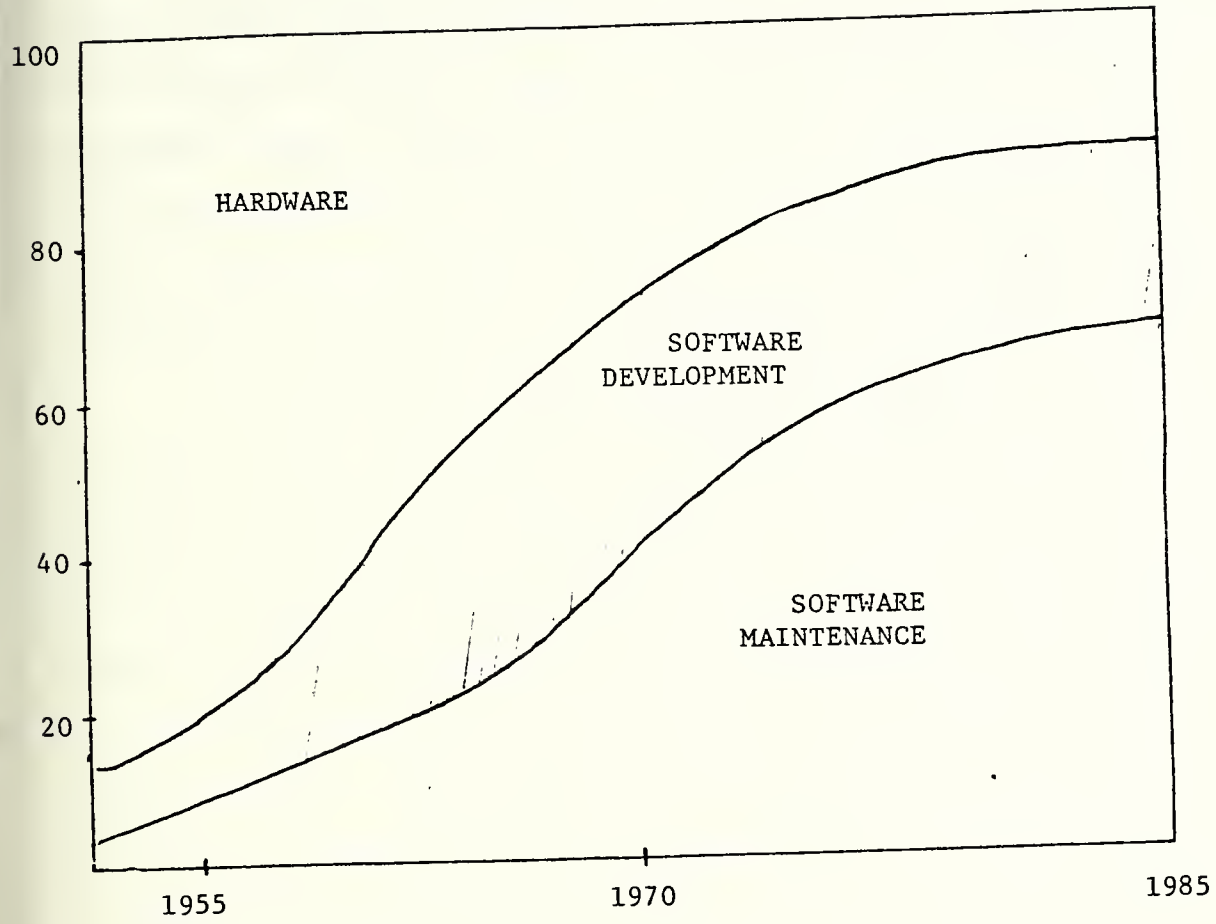


Figure 3. Software Maintenance Cost Growth

D. SOFTWARE DEVELOPMENT HISTORY AND CURRENT ISSUES

1. General

The seemingly limitless applications of computers forced an excessive demand on the productive capacity of the software industry from its inception. This demand for results and the disjointed response from many splinter companies and work groups created a chaotic, fragmented growth pattern. The sheer speed of growth precluded early development of stylized professional standards which could have aided individual project management control. Early development of such standards was defeated on at least three counts.

In the first place, there was and continues to be a perception that programming as an analytic activity conflicts with the intrusion of conventions and rules. At least in the minds of those involved. Many of the field's early successes required inspired, innovative, problem-solving approaches which might well have been stifled by the weight of ponderous standards. [12] The time proven bromide that standardization penalizes the best performances carries much credibility for those who participate in the analytic process.

Further, the traditional approach to programming involved much independent work and often formed a strong bond between individual programmers and their programs (which often symbolized a massive personal time commitment). Programmers thus tended to be somewhat irrationally blinded by pride of authorship when subjected to criticism of "their" program. This work environment was not conducive

to the development or imposition of universal programming standardization. [12]

Finally, as Paretta and Clark [13] point out, management control was deemphasized and thus ineffective in early software projects. The resulting dysfunctional behavior affected productivity and product quality and caused complexities to abound. Since programming ground rules were unknown, managers were often not able to distinguish many aspects of program quality (e.g., efficiency, maintainability, etc.). "Finding it difficult to reliably measure the quality dimension, quantity of output became the primary focus of control... The ability to keep programming projects on schedule, and to complete them on time thus became the two major criteria by which programmers were rewarded." [13] Despite these rewards, few projects came in on time with acceptable reliability. The natural response to such stimuli was a massive dose of suboptimization manifested by routine incorporation of shortcuts in software development. Such efforts focused on immediate tangible results to the detriment of long term consequences. Programs were patched together with focus on speed of completion and little or no interest in final structure or documentation. The proliferation of complex program structures in this environment is not surprising as the few planned structures that did exist were soon infested by layers of debug patches. Perhaps worst of all, an attitude of 'damn the documentation, full speed ahead' infused itself in the

profession's work habits to such a degree that it remains one of the most serious impediments to project control even today.

> "The pressure to produce working programs often meant that there was little time for programmers to think about documenting programs. The documentation that was available was mostly inadequate because few conventions existed for defining what should be included in program documentation, and for determining what level of detail was sufficient to make it comprehensible. Also, documentation was usually kept in the possession of the original programmer, and not in a program library where it could be made available for general use. This caused great confusion when one programmer was called upon to perform maintenance on a program written by another, especially when the latter was no longer with the firm." [13]

While the effects of much of this early confusion remain, a growing effort to identify and address such problems is evident.

2. Assessing Project Progress

Without doubt, the central historic issue in controlling software development has been the inability of management to successfully assess or predict progress in software development projects. [14, 15, 13] As noted earlier, the nature of software is characterized by the absence of physical characteristics. Since software development progress must be measured against a basically mental process of problem solving with no tangible outputs, early project managers often merely relied upon either questioning programmers or measure of man-hours expended to determine work

accomplished. Brooks [15] points out the folly of both practices. Individual programmers are universally over-optimistic with regard to evaluating their own work and abilities to complete a project quickly. Further, number of man-hours expended fails to measure the quality of time spent or the relative ability of those working together to effectively communicate and avoid redundant or conflicting activities. [15, 11]

To gain control, management must intelligently create intermediate deliverable items (e.g., specific design documentation) for which personnel can be held accountable. The quality (format, completeness, etc.) of deliverables can be specified by promulgated standards. Assignment and scheduling of resources to each of these deliverables constitutes the milestone approach to controlling development projects utilized by most organizations today. Management methodology used in resource estimates and allocations is still far from standard, often relying upon individual experience. Pioneering work in the principles of predicting resource requirements and tracking progress has been published but is not yet widely used. [e.g. 16, 17, 18]

3. Development Phase Interrelationships

Thibodeau and Dodson [19] postulate a cost prediction model which recognizes the impact of variable phase interrelationships on project utilization. In individual projects, these relationships may be either controllable or forced by constraints (of time, etc.). In either event, management

should be aware of their probable impact on schedule, performance and cost. While actual interrelationships are complex, the authors underscore the following project management issues:

- Inadequate resources allowed for design (and to a lesser extent coding) activities will result in more costly testing and/or higher error rates during life cycle maintenance.

- Planned phase overlaps (or deviations from the development plan that result in actual phase overlaps) adversely affect cost-driving variables.

- Software development activities are difficult to precisely define and restrict to particular phases--this ambiguity can be exploited in the process of cost reporting by inaccurately tying the easy to ascertain incurred costs to the more difficult to measure progress accomplished.

4. Quality Documentation and Configuration Management

In effect, software is documentation. The task of building another program copy from a full set of documentation would certainly be trivial compared to generating a replacement set of documentation solely from a program tape.

Further, quality documentation provides the following benefits:

- Assures full value and control of product when delivered to customer.

- Minimizes duplication of effort by recording solved problems.

- Saves interruption time by allowing future investigators to research on own.

- Compensates for the departure of an employee; consolidates work completed for the organization.

Paradoxically, while the importance of documentation is virtually unchallenged in the computer industry, the delivery of timely, complete and accurate documentation is rare. Much of this failure is attributable to the low esteem from which documentation suffers in the minds of most programmers. [12, 14] "The nature of programmers is such that interesting work gets done at the expense of dull work and documentation is dull work." [11]

Unfortunately, programmers must provide the bulk of effort in documenting their programs since they are the only available authority (without significant lead time). Therefore, management must provide an incentive and control structure that reinforces the importance of timely, quality documentation. This is best done with firm development standards to define milestone deliverables in detail, refusal by management to recognize development progress without delivery of appropriate documentation and the early institution of configuration management.

Configuration management is a control process which recognizes the importance of matching documentation with software and responds to the dichotomy between the ease of making program changes as opposed to the difficulty and tedium in making documentation changes. If program changes are allowed to be made without documentation, logical future program refinements or corrections will be impossible. "It is better

not to have any documentation than to have documentation of a former version. Without documentation it is at least clear that to modify the program reliably one should ... start from scratch." [20]

Configuration management is important throughout development but becomes critical in the integration and life cycle support phases when uncontrolled changes can ruin the entire project. When formal configuration control is in effect, each proposed code/documentation change must be submitted with justification and test plan (if applicable) for managerial approval. A properly run configuration control program will provide a developing organization the following benefits: [9]

- Software changes made in coordination with related hardware changes.
- Each software change appropriately tested and documented.
- Design new versions using existing software.
- If multiple versions are being maintained, ensure that corrections made to code are reflected in all common software.

5. Adequate Specification

Failure during a project's early stages to translate user requirements accurately and completely into both system and software specifications has been a major impediment to the success of many software developments. [21, 22] Inconsistencies and ambiguities introduced in this translation

process allow multiple intepretations during design and the inevitable accompanying complex structures which result when design guidance is allowed to convey variable meanings to those who implement. As a project progresses, conflicting development assumptions are often buried by short term efforts to force results by piecemeal patching aimed at satisfying piecemeal specifications. Residual inconsistencies and conflicts inevitably cause major problems (system degradations and failures) in integration/acceptance testing or during system operation, often with devastating consequences in additional resource commitment. Reluctance to produce formal, quality specifications stems largely from the level of effort and difficulty involved with their generation [23] and the propensity of projects to proceed on their own momentum by deriving requirements spontaneously as production needs dictate. Unfortunately, these requirements created 'on the fly' are often found to be in conflict with true user/customer desires. This result is not surprising since few customers plan thoroughly enough to know, in a project's early phases, exactly what they want, much less what words are required by analysts/programmers to guide production. The traditional result has been that specifications, which should function as precise bases for common agreement, often "abound with ambiguous terms ('suitable,' 'sufficient,' 'real time,' 'flexible') or precise-sounding terms with unspecified definitions ('optimum,' '99.9 percent reliable') which are potential seeds of dissension or lawsuits once the software is produced." [5]

Several difficult obstructions to management control ripple throughout a project if requirements specifications are of poor quality. Most visible of these is the positive growth in relative cost to correct errors during each succeeding development phase. Figure 4 [5, 24] depicts summary data from three corporations concerned with large scale software development. The wisdom of investing resources in a project to detect and correct errors in early phases such as definition/specification instead of relying on development/acceptance test efforts is evidently justified by quantum cuts in quality assurance expenses. Further, poor requirements specifications offer the following ills:

- User's inputs are minimized since no clear statement of desires exists.
- Management has no chance to exercise control since no clear production goals are available.
- No coherent guidance exists for design personnel.
- Test plans/procedures are impossible to write in good faith since there are no hard criteria for project performance available. [5]

Generating useful specifications is a time-consuming process for which the rewards of quality are normally not validated until the end of system development. This demotivating aspect has in great part accounted for the pitiful specification efforts that have crumbled beneath so many projects. Hope in this area has emerged in the form of growing attempts to automate the specification process. These efforts

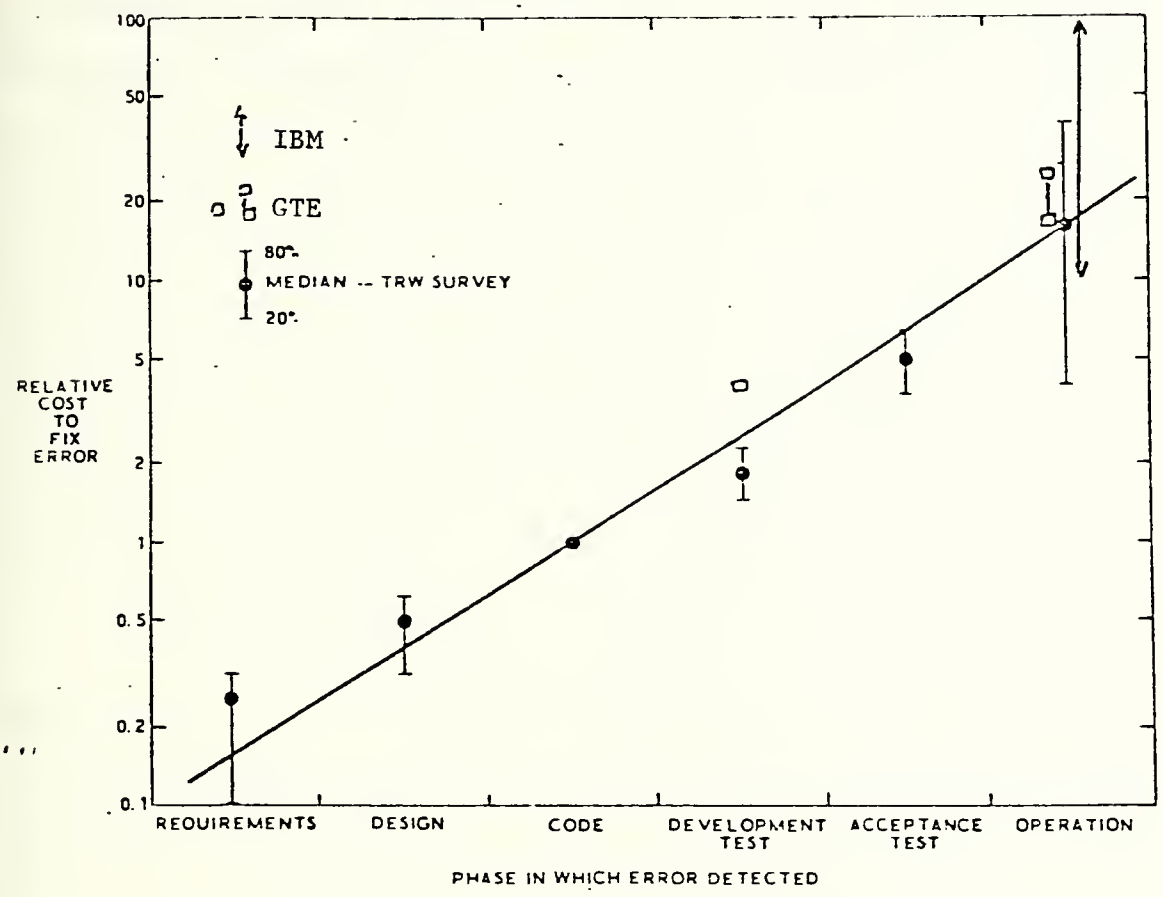


Figure 4. The Price of Procrastination

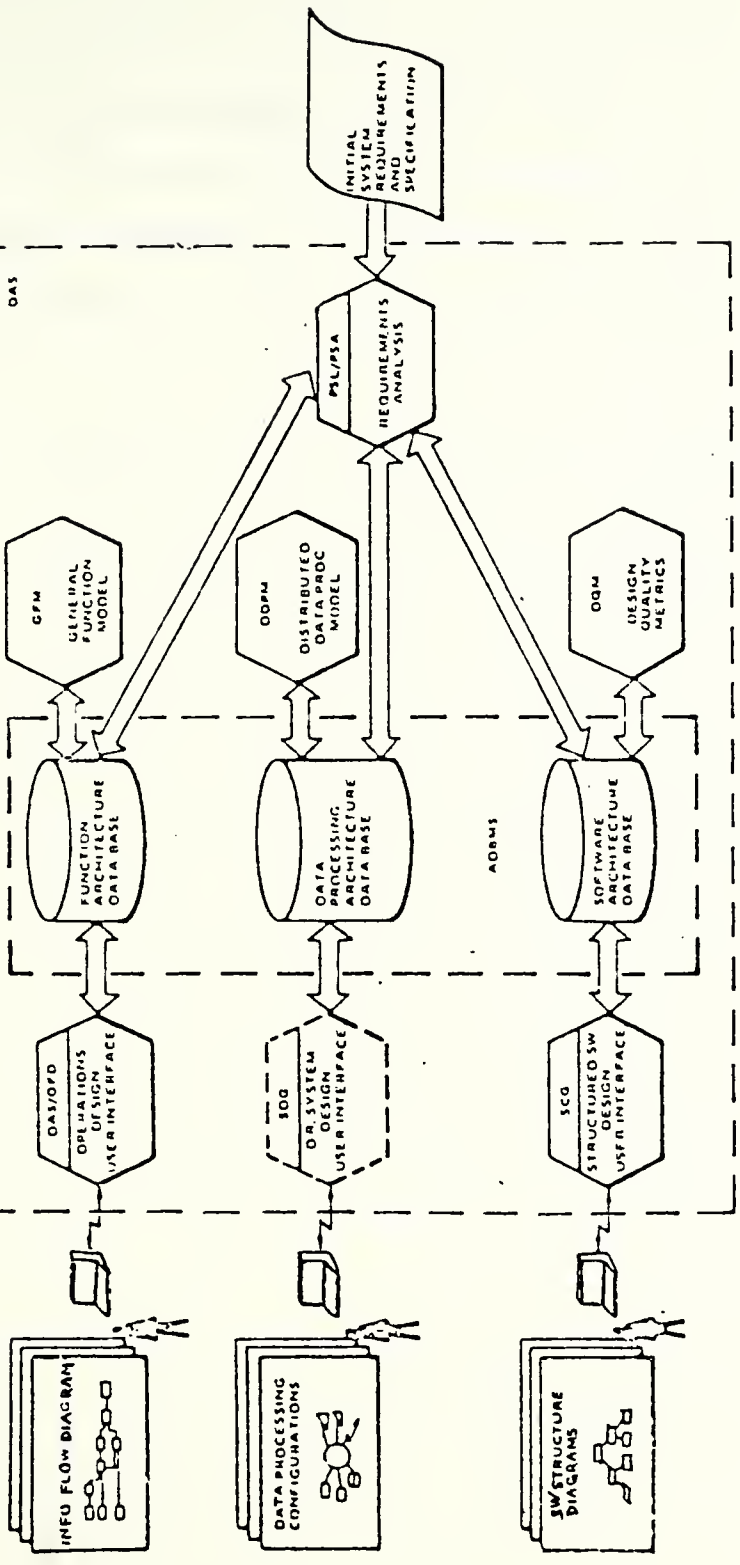
include Teichroew's work with PSL/PSA (problem statement language/problem statement analyzer), [25] Ross' Structured Analysis [26] and TRW's SREM. [24] With computer assistance, such systems are taking direct aim at eliminating or reducing the ambiguities, inconsistencies and omissions which have universally plagued specification generation. Their increasing use by development agencies is an encouraging indication of progress. TRW has developed and is continuing to perfect SREM. Variants of both SREM and PSA/PSL are under evaluation at FCDSSA. Hughes personnel have worked on a Design Analysis System (DAS) which incorporates PSL/PSA in an interactive, graphics oriented system supporting requirements, operations and software design verification. Figure 5 [27] depicts the innovative and ambitious DAS concept.

6. Top-Down Design

In the perfect project progression, all specification documents would be complete prior to the design phase. The design would then take form rather easily from precise specifications. For practical reasons, this is almost never the case. To feel comfortable with cost estimates, management has traditionally initiated one or more software designs prior to the continuance review at the completion of the specification effort. This rational demand for more information earlier is termed "The requirements/design dilemma" by Boehm [5] and is generally justifiable in the pursuit of improved estimation data. Unfortunately, this trend is often parlayed into a "bottom-up" approach to design wherein software

FIGURE 5

DESIGN ANALYSIS SYSTEM (DAS) CONCEPT



components are actually developed prior to appropriate consideration of potential interface and integration problems. Existing software components then drive the remaining design effort. This backward approach to design has survived for so long because there was little formal knowledge of the software design process or what makes a good software designer.

As in specification, design procedures developed spontaneously in the early 'cottage industry,' with no managerial guidance, are inadequate. The problem has been one of pressure to get on with the project and the result has often been incomplete designs which cause errors that are detected later in the project when cost to correct is highest. Reliability and life cycle costs suffer irreparably in this process. "More emphasis needs to be placed on software design so that the product is more reliable, less costly to maintain and easier and less costly to operate. So often, in the expediency of getting a product out of design, these factors are totally neglected to the later dismay of the user, when he discovers how much it costs to maintain and operate his new system." [10]

'Top-down' design, as practiced by a growing number of projects [23, 28, 30] seems to make much more sense in terms of projecting and maintaining control. "It begins with a top-level expression of a hierarchal control structure (often a top-level 'executive' routine controlling an 'input,' a 'process,' and an 'output' routine) and proceeds to iteratively

refine each successive lower-level component until the entire system is specified. The successive refinements, which may be considered as 'levels of abstraction' or 'virtual machines,' provide a number of advantages in improved understanding, communication and verification of complex designs." [5] If specification modifications are required in later stages due to changing user needs, disruptions will normally be minimized since corrections are restricted to lower-level code. (Higher-level code should require no modification as long as the major purpose of the program remains intact. [10]) Beyond the testimonial evidence of several completed projects, an indication of the labor/cost savings potential of top-down design was provided in initial experiments conducted by Comer and Halstead. [29] The product of an emphasis on complete, timely and quality design is the ability to focus early on the potentially most challenging project problem areas (e.g., interface definition and test strategies).

Previously mentioned automated specification techniques facilitate the top-down concept by providing "a medium for improved communication between the proponent (user), designer, coder and maintainer..." [30] Also of note are the efforts to improve design representation over the traditional flow charts. (E.g., The hierarchal input-process-output (HIPO) technique produces easy-to-understand graphics which represent software in a hierarchy of modules, each of which is symbolized by its input, its output and a summary of the connective processing. [5, 3])

7. Ordered Program Structures

The historic causes (lack of standardization, pressures to produce quickly, inadequate documentation, etc.) and resultant ills of complex coding structures have been mentioned. Several techniques have been proposed and utilized to simplify these structures.

a. Structured Coding (or Structured Programming)

The theory of structured coding, developed by Dijkstra [32] and expanded into a set of techniques by him and others, is now in widespread use (including at the three facilities visited: TRW, Hughes and FCDSSA). The most significant feature of structured coding is the recognition that an excess of branching statements contributes enormously to structural complexity. With this realization in mind, program modules are limited to single points of entry and exit and branching statements within modules are strictly controlled. Following these techniques maximizes sequential logic flow and contributes greatly to readability and the enhancement of all revision quality factors by simplifying and standardizing program constructs.

b. Chief Programmer Team

The Chief Programmer Team (CPT) concept [33, 34] is analogous to a surgeon surrounded by a staff of specialists whose function it is to maximize his performance. The chief programmer similarly acts as an expert surrounded by programmers who improve his efficiency by accomplishing all routine tasks and free the expert to concentrate on the most difficult aspects of the project.

"The chief programmer team represents a managerial approach to program development that offers some needed relief from the problems of organizational structure... The emphasis in chief programmer teams is on producing programs that are well designed by taking advantage of experienced programming talent, rather than delegating important programming functions to inexperienced programmers on a 'sink or swim' basis. Because the team is organized around experienced programmers, projects can develop more quickly and with more direction than when conventional staffing approaches are used. Instead of just being part of a poorly led thundering herd of junior programmers, each member of the team is a specialist who makes an individual contribution to the project under the close direction of the chief programmer. The arrangement enables better utilization of personnel, reducing the number of people involved in a programming project. Not only does this generate immediate cost savings, it also suppresses the numerous communication and coordination problems so often associated with software projects. As an active participant in all stages of development, the chief programmer is also in a better position to evaluate the headway the team is making on a project. His direct involvement means he does not have to rely on tangible evidence to gauge a project's progress." [13]

A modified implementation of CPT by Naval Air Development Center, Warminster for the CVTSC software project noted positive results in maintaining design consistency and minimizing integration problems "which arise from conflicting implementations." [35] On the negative side, this approach may be limited by the manning available. Further, it is doubtful that a career programmer will desire to spend more than a few projects functioning at the absolute direction of the "Chief Programmer."

c. Program Walkthrus - Egoless Programming

Weinberg [12] articulated the problems created by ego involvement of programmers with any code that they produce. "A programmer who truly sees his program as an extension of his own ego is not going to be trying to find all the errors

in that program. On the contrary, he is going to be trying to prove the program is correct--even if this means the oversight of errors which are monstrous to another eye." To combat this blinding and destructive link between programmers and their code, "program walkthrus" have been instituted. In this technique a review group of the programmer's peers (i.e., no management personnel) scrutinize code in detail prior to running it on a computer in order to detect errors as early as possible. Key to such proceedings is the atmosphere of correcting 'our' product and never of attacking 'your' programming skill. Reviewing/presenting roles must be rotated to avoid pressure build-up from constant review. [13]

8. Test/Integration

Testing and debugging large scale software remains the most tedious, frustrating, expensive and unpredictable phase of development. Despite massive expenditures, testing successes remain limited, by virtue of the overwhelming size and complexity of many large scale systems. Operation software is never completely free from error. Proof of the ineffectiveness of past and current testing techniques are the inevitable residual errors that occur after the most rigorous testing available: (e.g., "Software systems used for the Apollo manned spaceflight program are probably one of the most thoroughly tested programs in the world. Yet software failures were detected in Apollos 8, 11 and 14." [36])

As in specification and design, initial industry attempts to predict/guarantee satisfactory software operation

(i.e., reliability) were somewhat misguided. Specifically, the differences between the well understood engineering principles regarding hardware failure and repair and the phenomenology of software errors and correction were not fully appreciated. The result of these differences was a general misapplication of assumptions concerning required level and type of test effort for software products. The effects of these misunderstandings are manifested in the dramatic increase in the ratio of actual to predicted costs to maintain programs--i.e., to correct designs and debug residual errors remaining in operational software after satisfactory completion of testing. (Figure 3 [5] depicts this growth. Note: Maintenance costs also include update design changes.)

Analysis of the growing body of data concerning software errors is now providing a number of germane insights into their nature which should be closely considered in future projects. [5] These insights include the following:

- Program complexity is a major factor in the propensity of making programming errors and the level of effort required to detect and correct. [1, 37]
- The development of test plans should begin as soon as possible after specification. This early development can pinpoint inconsistencies and omissions in the software specifications. [9]
- Testability should be an important consideration in program design and architecture. [38]

- Specification should be accomplished with potential structural complexity and ease of testing as prime considerations. [39, 40]
- A series of automated aids for test generation and program evaluation under current development or appraisal have shown excellent potential for improving program quality and reducing development costs. [36]

9. Verification and Validation (V&V) [6]

Concern for assuring quality in large scale programs has led to the development of a systematic process of analyzing and testing documentation and code. This process takes its name from its two aims:

Verification - The determination that each development phase satisfies formal and logical requirements of preceding phases.

Validation - The determination that the developed software and documentation satisfies all performance requirements.

(The term validation is used in several different senses in the somewhat related fields of Department of Defense (DOD) system/software acquisition and software development. These differences should be understood.

- A requirements 'validation' activity occurs in the first (conceptual) phase of DOD system acquisition. This activity addresses the legitimacy of defined requirements to satisfy stated needs.

- The second phase of DOD system acquisition is termed the 'Validation Phase.' Here 'validation' refers to the conceptual proof that the solution (e.g., preliminary system design) is ready to proceed into full scale development. [7]

- As used in 'V & V,' 'validation' is a set of activities that occur during the test and integration phase of software development.)

A properly implemented V & V program, invoked in a project's earliest stages, can both assure software quality and aid management in assessing development progress. As in all quality assurance activities, the program must be accomplished by a technically competent, independent team having no political connections with the development group. [40, 41] Specific techniques utilized by the V & V team can be adapted to the particular program characteristics (real or non-real time, scientific or business, algorithmic or logic intensive) and depend upon a case-by-case cost-effectiveness determination. Information derived is useless unless fed back for timely management review and utilized to key iterative improvements to deficient areas. A general chronological list of objectives and possible techniques is included below:

- Requirements Verification - Analyze each requirement for criticality, risk, testability, and impact on software.
- Set up mechanism to assure traceability.

- May use problem statement languages, correctness proofs, truth table exercises, abstract simulations.

— Design Verification - Examine design logic, structure, data base design, architecture and documentation considering impact on all revision, operation and transition quality factors. (Correctness, efficiency and usability are emphasized.)

- May use special design languages, analytic techniques, special simulators and models. [41]

— Code Verification - Much iteration between Design and Code efforts expected.

- Inspect code to ensure design goals are followed, complex structures minimized, organization's procedures followed.

- May use inspection, automated analysis aids (e.g., static/dynamic analyzers, standards enforcers, data base verifiers), automated tracing mechanism, emulators, code level simulators.

— Validation - Parallels test and evaluation.

- Includes both monitoring of developer's test efforts and independent tests.

- All quality factors important but emphasis is on correctness and reliability.

- Continuing thread of traceability from requirement to design to code to test is a key.

The V & V concept is enjoying increased visibility and implementation, particularly in DOD-related projects. Whether or not its nomenclature is formally used in all or part of an individual quality control program, quality assurance goals and limitations remain the same. Quality is a function of the complete development cycle and cannot be tested or monitored into a system. A rigorous review and audit function is only as good as the effectiveness of its feedback loop in causing timely product and process improvements. [6] The potential of a quality assurance organization's success is thus defined by the extent of promotion and backing it receives from management policy and action.

III. COMPLEXITY

A. GENERAL

The essence and impact of complexity as it relates to computer programming is a difficult concept to convey and quantify. Despite the difficulty, widespread recognition that a better understanding of this relationship will doubtless lead to improved management and an accompanying reduction of software development costs has stimulated a growing descriptive effort in the literature. In this chapter, an attempt will be made to consolidate and extend the major thrust of these ideas.

The traditional concepts--extent of varietal content and degree of interrelationship--continue to be germane. However, difficulties have arisen in applying these concepts to system and software assessment and management. The description of a particular aspect of complexity is often accompanied by a metric--i.e., a method of qualification (by measuring a surrogate) designed to provide an indication of the extent of complexity present in a problem-solving process, computer program or system. When a particular metric is heavily used in a production or research project, language often becomes relaxed and the distinction is sometimes lost between the abstract degree of complexity present and the explicit attempt to measure one of its manifestations. Since a potential for false indication exists with all surrogate measures, this distinction should be considered in the interpretation of

each metric. The most trivial complexity metric is merely the number of source statements present in a program. Although this metric is hardly accurate alone, large programs are generally more complex than smaller ones and size is sometimes useful in gauging the meaning of other metrics.

The following section lists a number of methods devised to classify/quantify various facets of complexity.

B. TYPES OF COMPLEXITY

1. Conceptual and Software Complexity

Conceptual complexity refers to the level of difficulty associated with conceiving and solving the real world problem. Software complexity covers the form and structure that results when this solution is translated to a computer language. While these two aspects are not independent, their functional relationship is neither simple nor consistent. Indeed, the most trivial of concepts can be transformed into a computer program so complex as to confound all efforts to trace logic flows, find errors or make minor modifications. Conceptual complexity is important to project management and must be considered appropriately in terms of manpower mix, etc. However, it is the inability to understand and control software complexity which has traditionally been the downfall of major projects. Classifications of computer related complexity have generally either attempted to clarify or to further subdivide conceptual and software complexity.

2. Computational Complexity

Computational complexity is the level of involvement and difficulty associated with computing functions. [42] Work in this field deals with quantitative aspects of computed solutions, recursive function theory and analyses of computational models like the Turing machine. [e.g. 43, 44] In relation to computer programs, computational complexity metrics generally provide data relevant to some program resource usage. CPU run time and core usage were among the first concerns of programmers and directed early attention to these manifestations of complexity. (As multiprocessed and time shared computer systems evolved, other measures (e.g., channel usage, device usage, secondary storage requirements, supervisor usage, etc.) became important considerations. [45] While these usage measures are related to complexity, they are generally not considered direct manifestations.)

In describing computational complexity in logic circuits and the Turing machine, Savage [46] identifies the following complexity measures:

- Computational complexity: a measure of the 'size' of a logic circuit. "The combinational complexity of a function f relative to a basis Ω (set of Boolean functions such as AND, OR & NOT), denoted $C_{\Omega}(f)$, is the minimum number of elements from Ω needed to realize f with a logic circuit."

- Delay complexity: a measure of the 'depth' of a logic circuit. The depth of a combinational machine is "equal to the number of logic elements on the longest (directed) path from inputs to outputs. The delay complexity of f with respect to Ω is the depth of the smallest depth circuit over Ω for f."
- Turing machine program complexity: the length of shortest length program for a function : f: $\{0,1\}^n \rightarrow \{0,1\}^m$ on a Turing machine.

3. Psychological Complexity

Psychological complexity concerns characteristics of an individual program which make it difficult to understand and manipulate. "...psychological complexity assesses human performance on programming tasks." [42]

4. Subjective Metrics

In the early phases of a project, predicted complexity must be based upon the subjective evaluations of early planners. Many organizations rely almost wholly upon prediction by experienced analysts and programmers for cost estimates and follow-on planning data. Such predictions take into account similarities and differences with past projects and naturally vary from individual to individual or group to group. Subjective complexity ratings may simply be expressed by quality descriptors (e.g., 'extremely complex,' 'very complex,' etc.) or may be translated to rank numbers (etc. from 1 to 5), depending upon the requirements stated by managers. While such approaches may be useful in preliminary cost estimates

when little time or concrete data is available, their lack of precision and susceptibility to individual bias generally make them unacceptable for detailed planning, resource allocation or test strategy formulation. Despite these apparent weaknesses, many organizations have yet to progress beyond subjective appraisals of complexity.

5. Gilb Metrics

Gilb [47] proposed a methodology to measure and compare logical and structural aspects of complexity in various systems. [48]

- Logical complexity: the extent of decision-making logic within a program or system. The metric considers "absolute logical complexity" (C_L = number of nonnormal exits from a decision statement (IF, ON, AT END, etc.) and "relative logical complexity" (c_L = ratio of C_L to total number of instructions).
- Structural complexity: degree of interrelationships between subprograms or subsystems. The metric considers "absolute structural complexity" (C_S = number of modules or subsystems) and "relative structural complexity" (c_S = ratio of module/subsystem linkages to the total number of modules/subsystems).

6. Thayer Complexity Model

Thayer [49] offers consideration of various measurable complexity surrogates, both separately and together (via

weighted formula), to understand the error proneness and probable difficulty of error detection and correction in a program. [48]

- Logic complexity metric (referred to as Total Logic Complexity, L_{TOT}) can be numerically evaluated for each routine by calculating:

$$L_{TOT} = LS/EX + L_{LOOP} + L_{IF} + L_{BR} ,$$

where

LS = number of logic statements

EX = number of executable statements

L_{LOOP} = computed loop complexity for the routine in accordance with the following equation (values scaled by x 1000):

$$L_{LOOP} = \Sigma m_i W_i ,$$

where

$$W_i = 4^{i-1} \frac{3}{4^Q - 1} \quad \text{so that} \quad \sum_{i=1}^{Q} W_i = 1 ,$$

and

m_i = number of loops in routine at indenture or nesting level i

W_i = weighting factor

Q = maximum level of indentures in the system

4 = shaping value .

L_{IF} = computed IF complexity (number of IF statements, nesting level) in accordance to the following equation (values scaled by x 1000):

$$L_{IF} = \Sigma n_i W_i ,$$

where

n_i = number of IFs in routine at indenture or nesting level i

W_i = weighting factor

L_{BR} = number of branches BR, times 0.001 .

- Interface complexity metric (C_{INF}) can be numerically evaluated by calculating:

$$C_{INF} = AP = 0.5\ (SYS)\ ,$$

where

AP = number of application program interfaces

SYS = number of system program interfaces

0.5 = estimated interface weighting factor.

- Computational complexity metric (CC) can be numerically evaluated as follows:

$$CC = (CS/EX)\cdot(L_{SYS}/\Sigma CS)\cdot CS\ ,$$

where

CS = number of computational statements

L_{SYS} = ΣL_{TOT}, (total logic complexity for each routine)

CS = the sum over all routines of the values of CS for each routine .

- Input/output complexity metric ($C_{I/O}$) is defined for each routine as follows:

$$C_{IO} = (S_{I/O}/EX)\cdot(L_{SYS}/\Sigma S_{I/O})\cdot S_{I/O}\ ,$$

where

$S_{I/O}$ = number of input/output statements

$\Sigma S_{I/O}$ = sum over all routines of the values of $S_{I/O}$ for each routine.

- Readability (U_{READ}) is defined for each routine as follows:

$$U_{READ} = COM/(TS + COM) ,$$

where

TS = total number of statements (executable plus nonexecutable, exclusive of comment statements)

COM = number of comment statements.

- Total complexity (C_{TOT}) combines all factors as follows:

$$C_{TOT} = L_{TOT} + 0.1C_{INF} + 0.2CC + 0.4C_{I/O} + (-0.1)U_{READ}$$

7. McCabe Graph - Theoretic Complexity Model [50, 51, 48]

a. Graph Model of Programs

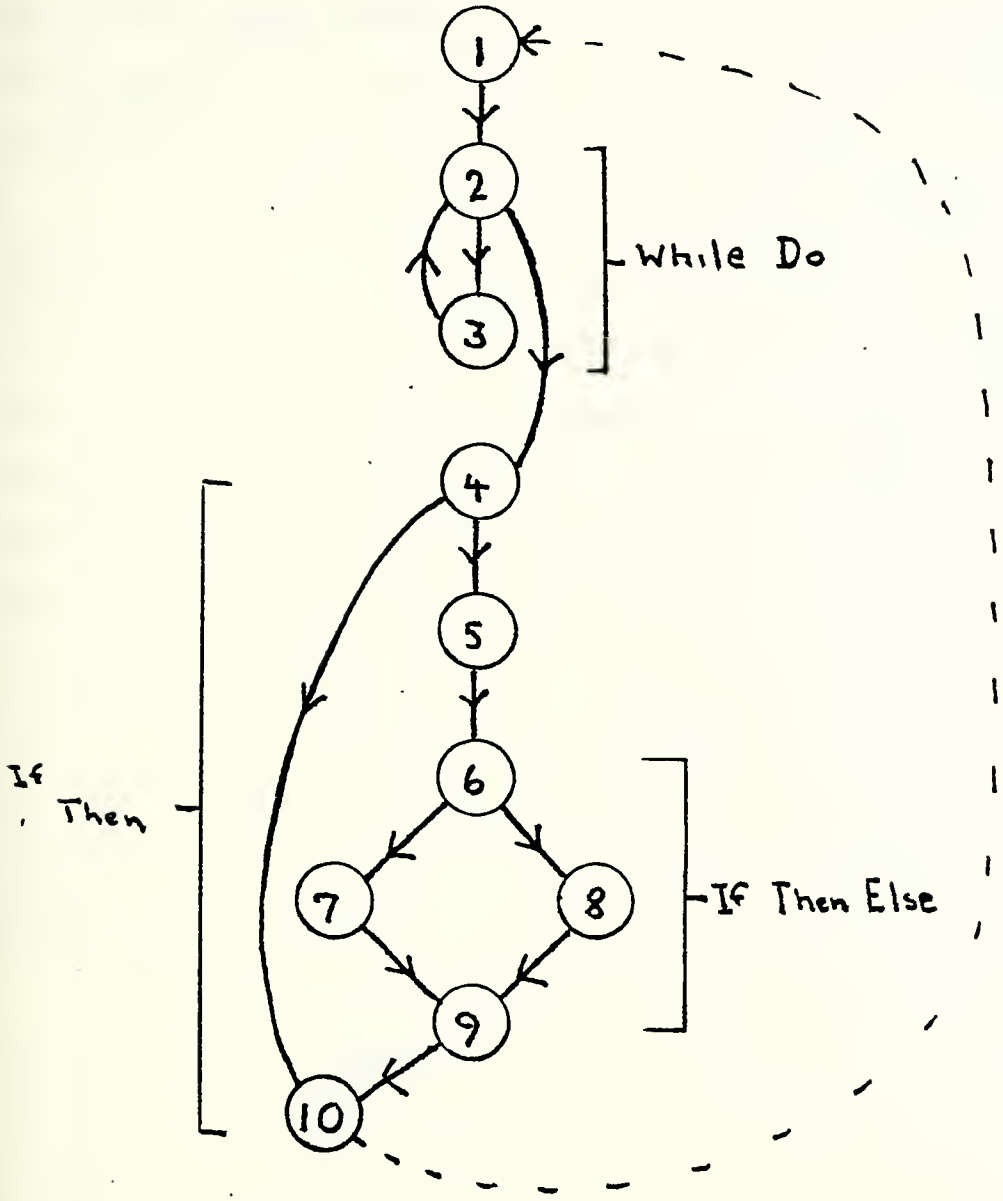
McCabe [50, 51], Schneidewind [1] and others have noted the validity of utilizing the graph model to represent computer program structure. Briefly defined, a graph of a program is composed of a set of nodes connected by a set of directed arcs. The nodes represent statements or elements of a program while arcs represent program control flow.

Figure 6 [1] shows a graphic representation of a simple program which includes several basic program constructs. In analyzing control flow from a given node, 'successor' or 'predecessor' nodes are determined by the indicated directions of connecting flow. [45]

The most significant benefit of the directed graph model is the attendant ability to measure certain complexity surrogates related to the graphic representation. These measures can then be used to control complexity and develop optimal testing methodology.

FIGURE 6

DIRECTED GRAPH REPRESENTATION OF A SIMPLE PROGRAM



b. Cyclomatic Complexity Metric

McCabe [50] defines cyclomatic number V of graph G with n vertices, e edges and p connected components as follows:

$$V(G) = e - n = p$$

By limiting application of this definition to single entry and exit programs, an equivalence between V and the maximum number of linearly independent circuits is asserted.

Schneidewind [1] extends this interpretation as follows:

"Since V is equal to the number of independent circuits, it is equal to a set of sub-structures which can be identified in a directed graph. When structured programming techniques are used, the independent circuits are identified with the constructs of structured programming: While DO, IF THEN, IF THEN ELSE, etc." Further, "...by generating all circuits from the fundamental circuits, the different execution sequences which must be tested can be identified. Secondly, the frequency of occurrence of an arc in the circuits indicates the relative importance of testing the arc."

c. Other Directed Graph Related Complexity Metrics

- Reachability (R): summation, over the nodes, of the number of available ways to reach a node. (Average reachability (r) = R/# of nodes.)
- Number of Paths (N_p): minimum number of paths (i.e., no loop traversed more than once in succession).

- Number of Nodes (N_N)
- Number of Arcs (N_A)

8. Halstead Metric

Halstead [e.g. 52] proposed and refined a comprehensive discipline concerning "measurable properties of written material expressed either in computer program or in prose." [53] The chief tenet of this discipline (now known as Software Science) is the application of natural science methodology to investigate characteristics of written communication. With regard to software complexity, Halstead reported an important metric to gauge program difficulty which took into account the variety of instructions (vocabulary) and their frequency of usage (length). Instructions were subdivided by operator codes and operand addresses. The Halstead effort metric (E) is calculated as follows:

$$E = \frac{\eta_1 N_2 (N_1+N_2) \log(\eta_1+\eta_2)}{2\eta_2} ,$$

where

η_1 = number of unique operators

η_2 = number of unique operands

N_1 = total frequency of operators

N_2 = total frequency of operands .

This value indicates the number of mental comparisons required to generate a program. Follow-up experimental work has found significant correlation between Halstead's metrics and such measures of programmer performance as program errors, program quality and time to program. [54]

9. System Complexity

Much of the theory developed around aspects of conceptual and software complexity can be abstracted and applied to the organization and structure of systems. As an example, the directed graph model might be utilized to represent a system structure with communication paths translated into arcs and modules translated into nodes. A cyclomatic number analysis can then be used to indicate the more complex system structures and/or used in the system design process to maintain ordered system structure.

IV. THE NAVAL POSTGRADUATE SCHOOL (NPS) EXPERIMENT

A. PURPOSE

As indicated in the previous chapter, numerous theoretic approaches to defining and measuring complexity have been proposed. While these approaches are useful in understanding complexity relative to the programming task, many of them are difficult to apply directly to management control, either because they are too subjective (e.g., psychological complexity), because they require data that is unavailable until the project is essentially complete (e.g., the Halstead Metric) or because they have not yet been sufficiently corroborated by empirical data. In an important step to address this operational requirement, Schneidewind [1] directed an experiment conducted by Hoffman at the Naval Postgraduate School (NPS) designed to provide quantitative data in support of the following:

- The hypothesis that complexity is a significant determinant of both the propensity to commit programming errors and the time required to detect and correct existing errors.
- If the hypothesis is true, a determination of valid complexity measure(s) to predict probability of programming error commission and the difficulty of error detection/correction.

Detailed methodology and results of this experiment are available [1, 37] and will not be covered here. However, for continuity of discussion, a brief overview of the experiment and its potential application is presented.

B. APPROACH AND RESULTS

Corroboration and extensions of the previously cited work by McCabe [50] concerning cyclomatic numbers and other measures were primary concerns of the NPS experiment. In conducting the experiment, four projects were programmed by Hoffman as part of his Masters in Computer Science Degree requirements. [37] The work was accomplished in ALGOL W for IBM 360/370 execution. Such software engineering concepts as top-down design and structured walkthrus were used throughout. Error categories were broken down in comprehensive detail. Information was then collected concerning the design, coding, debugging and testing phases of each project along with error listings recording the nature of each error discovered. Of particular interest was the distribution of labor time used to detect and correct errors and the relation of selected complexity metrics to the structure containing each error. Table III [1] depicts project sizes and man-hour distribution. The following complexity metrics were evaluated:

- NUMBER OF PATHS (N_p)
- CYCLOMATIC NUMBER (V)
- REACHABILITY (R)
- AVERAGE REACHABILITY (r)
- NUMBER OF SOURCE STATEMENTS (S)

TABLE III

Project Characteristics

PROJECT TYPE	NO. OF SOURCE STATEMENTS	MAN-HOURS					OPERATING SYSTEM
		PROGRAM DESIGN	CODING	DEBUGGING	TESTING	TOTAL	
String Processing	141	5.0	7.0	4.0	5.8	21.8	OS/MVT
Directed Graph Analysis	712	31.0	26.0	55.0	13.0	125.0	OD/MVT and CP/CMS
Addition* to Project 2	70	7.0	4.0	3.0	19.0	33.0	OS/MVT and CP/CMS
Data Base	1084	24.0	24.5	41.5	11.0	101.0	CP/CMS

* Addition to Reachability and Reachability Index computations to Directed Graph Program.

Results and analyses indicated that while a linear relationship could not be proved, all complexity metrics considered were significantly higher for structures which had errors, thus supporting the original thesis. Tables IV and V [1] summarize these results. Also, error detection and correction times were generally longer for programs of higher complexity metrics. Further:

> "When the number of errors found in procedures was correlated with cyclomatic number and number of source statements, the correlation coefficients were higher for other complexity measures. It also appeared that these two measures were related to the total error detection and total error correction times. It was learned that trying to keep the cyclomatic number small not only reduced the number of errors but also contributed to the reduction of debugging and testing efforts." [37]

C. PROJECT SCOPE

Two limitations of scope should be recognized in evaluating results of the NPS experiment:

- Designing and Coding/Debugging activities were emphasized at the expense of analysis and integration issues.
- The small scale of the projects raises the question of validity in extrapolating conclusions directly to large scale software development projects.

D. APPLICATION

The major value of the NPS experiment is the high quality of error data obtained in terms of detailed error type definition and careful recording procedures. Reported results provide an important corroboration of McCabe's work, strongly

TABLE IV

NPS EXPERIMENT

Correlation Coefficients
(Error Properties vs. Complexity Measures)

Number of Errors Found vs.		Number of Procedures
Cyclomatic Number	.78	31
Number of Source Statements	.59	31
Number of Paths	.76	20
Reachability	.77	20
Average Reachability	.78	
Labor Time (Man-Mins) to Find Error vs.		
Cyclomatic Number	.67	31
Number of Source Statements	.59	31
Number of Paths	.90	20
Reachability	.90	20
Average Reachability	.87	20
Labor Time (Man-Mins) to Correct Error vs.		
Cyclomatic Number	.72	31
Number of Source Statements	.51	31
Number of Paths	.65	20
Reachability	.66	20
Average Reachability	.71	20

TABLE V

NPS EXPERIMENT

Complexity Measure Comparison
(Procedures with no Errors vs. Procedures with Errors)

	No Errors		Errors	
	Mean Value	Number of Procedures	Mean Value	Number of Procedures
Cyclomatic Number	1.699	83	4.74	31
Number of Source Statements	9.361	83	27.23	31
Number of Paths	2.671	82	27.1	20
Reachability	10.1	82	120.3	20

indicating that "it would be worthwhile to use complexity measures as a program design control to discourage complex programs and as a guide for allocating testing resources." [1]

V. THE ROLE OF COMPLEXITY IN RESOURCE ESTIMATION AND ALLOCATION

A. GENERAL

It can be argued that blame for the historically inaccurate cost predictions indicated in Chapter I can be attributed to poor estimation techniques as well as to the management control issues emphasized in Chapter II. Widespread acknowledgement of this failing is reflected by the impressive extent of research and experimentation in the past decade directed to improve the largely judgmental state of the art that persists in software cost estimation. This chapter will briefly cover certain problems and approaches involved in the estimation process, describe and offer an evaluation of one existing model (Putnam) and suggest an application of the cyclomatic number complexity metric to resource estimation and allocation.

B. ISSUES IN SOFTWARE RESOURCE ESTIMATION

1. New Dynamic Field

Wolverton [11] observes that "the software industry is young, growing, and marked by rapid changes in technology and application. It is not surprising then, that the ability to estimate costs is still relatively undeveloped." Beyond the significant number of evolutionary improvements to the programming profession wrought by its practitioners, the direction of the software development process has largely been

driven by the frantic rate of new developments in computer hardware which were generally not aimed at rectifying software difficulties. This dynamic, disjointed environment of change has prevented the development of mature cost estimation techniques.

2. Quality and Testing

One important manifestation of the changing nature of software development is the growing emphasis toward testing to assure quality. As quality becomes more an issue in system design, the proportionate amount of time spent in each phase of the development cycle will change, thus invalidating past project guidelines and making estimation more difficult. [54]

3. Programming Units of Measure

Wolverton [55] cites the unreliability of available units of measure used to gauge programming quality and productivity as one of the most difficult impediments to accurate software cost estimation (as well as software management). His list of measures which can produce false indications in certain circumstances includes the following:

- Lines of code written per programmer month.
- Man months of effort per k lines of code.
- Defects per k lines of code.
- Man months of effort per k bytes of code.
- Object instructions measurements.
- Man hours per instruction.
- Cost per defect.
- Defect removal per k lines of code.

- Defects processed per man month.
- Machine hours and terminal hours used per programmer month.
- Machine hours and terminal hours per k lines of code.
- Cost per page of documentation.

4. Fragmented and Proprietary Research

While the academic orientation of the programming profession has encouraged and supported publication of much of the detail pertaining to newly devised estimation procedures, actual large scale projects are almost totally accomplished by individual firms in a competitive industry. Protective policies and the mechanics of responding to requests for proposals have placed much empirical data from specific projects in a proprietary category. Thus the important experimental data from individual project failures and successes in different firms has not been comprehensively assimilated.

5. Individual Resource Costs

a. Labor

The labor factor of software development cost is highly dependent on programmer productivity. Unfortunately for estimation efforts, individual variances in productivity are extreme and difficult to predict. As an example, Ogdin (quoted in 56) cites a study involving twelve experienced programmers who accomplished the identical programming task with the following productivity variances:

-25:1 in coding time

-26:1 in debug time

-11:1 in CPU time

-13:1 in execution speed

- 5:1 in number of lines coded.

The existence of these wide performance variances causes such difficulty in conducting controlled experiments regarding the utility of programming languages, tools and techniques that productivity fluctuations often shield the influence of the factor under investigation. Productivity rates of a specific individual or group in a particular internal environment must be appropriately assessed if cost estimates are to be accurate.

b. Elapsed Time

The amount of calendar time available for a software development project has a significant impact on costs. A useful cost estimate must provide guidelines for the allocation of resources over the total predicted elapsed time to accomplish the following:

- Coordinate time-phased funding.
- Account for costs that are time dependent.
- Assign resources for all explicit and implied tasks resulting from the work unit breakdown.
- Manage the project within budget constraints.

It is apparently critical that management appreciate the time requirements of a prospective project early in the estimation/bid process. While schedules are normally specified in development contracts, development organizations must approach original acceptance of contract schedules or later

schedule changes with the utmost caution and a thorough risk analysis. [57] As Brooks points out:

> "The number of months of a project depends upon its sequential constraints. The maximum number of men depends upon the number of independent subtasks. From these two quantities one can derive schedules using few men and more months. (The only risk is product obsolescence.) One cannot, however, get workable schedules using more men and fewer months. More software projects have gone awry for lack of calendar time than for all other causes combined." [15]

c. CPU Time

In the past, a difficult management issue to resolve was the appropriate trade-off to be made between slack computer time and slack programmer time. In one case, if computer time was so scarce that programmers could not be guaranteed access to a machine, progress was held up and schedules degraded. Alternately, if computer time was easily available with few effective constraints, programmers tended to attempt much of their analysis, design and debug work on the machine when another environment might have been more suitable and efficient. [57] With the current availability of interactive terminals and sophisticated software test tools, coupled with the high cost of programming labor, management's role appears to have been altered to one of ensuring availability of appropriate tools and work environment to maximize productivity.

6. Lack of Sufficient Software Engineering Data Base

Boehm [58] explains the difficulty involved with analyzing software problems thoroughly as follows:

> "One of the reasons progress has been so slow is that it's just plain difficult to collect good software data... These difficulties include:

- Deciding which of the thousands of possibilities to measure.
- Establishing standard definitions for "error," "test phase," etc.
- Establishing development performance criteria.
- Assessing subjective inputs such as "degree of difficulty." "programmer expertise," etc.
- Assessing the occurrence of post facto data.
- Reconciling the sets of data collected in differently defined categories.

7. Continuous Project Change

An individual engaged in cost estimation must live with the fact that the program being estimated is never the program actually developed. Changes may occur as the result of the user finally discovering what he really wants, the developer finally owning up to his inability to solve the technical problem or an unforeseen change in the environment. Whatever the reason, the change process has been observed so frequently that Lehman has pronounced its inevitability as his "First Law in Large-Program Evolution."

> "The Law of Continuing Change arises from the fact that the world, in this case the computing environment, undergoes continuing change; all programs are models of some part, aspect or process of the world. They must therefore be changed to keep pace with the needs of a changing environment, or become progressively less relevant, less useful and less cost effective." [59]

8. Documentation

Software documentation constitutes one of the largest and most difficult to manage 'hidden' costs in software

development. When it is contracted for and produced in quantity, it is normally not adequately reviewed and rarely fulfills its functions. Alternately, when it is minimized as a cost saving measure, both the customer and the developer (not necessarily in equal proportions) suffer future costs in reinventing solutions. Figure 7 [60] shows the theoretical relationship of varying documentation costs to total project costs with a hypothetical optimum documentation level. [61]

9. Ability to Transfer Existing Code

An important opportunity to save development costs obviously exists when part of the programming has been accomplished previously. Cost estimates vary according to the amount of project code that must be newly generated or can be transferred or retrofitted from existing programs. However, estimates involving transfer and retrofit of code are unique problems which must take into account required interfaces and design constraints required to make existing code fit. Forcing existing code into a design may result in unwanted complex structures. At some point a developer may find it more cost effective to rewrite code than to transfer or retrofit it. This evaluation should be an output of the estimation process. [57]

C. TYPES OF ESTIMATION

In this section the major approaches to estimation are categorized and briefly described. It should be noted that in practice more than one approach is frequently used, either in combination or as cross verification, while evaluating a single project.

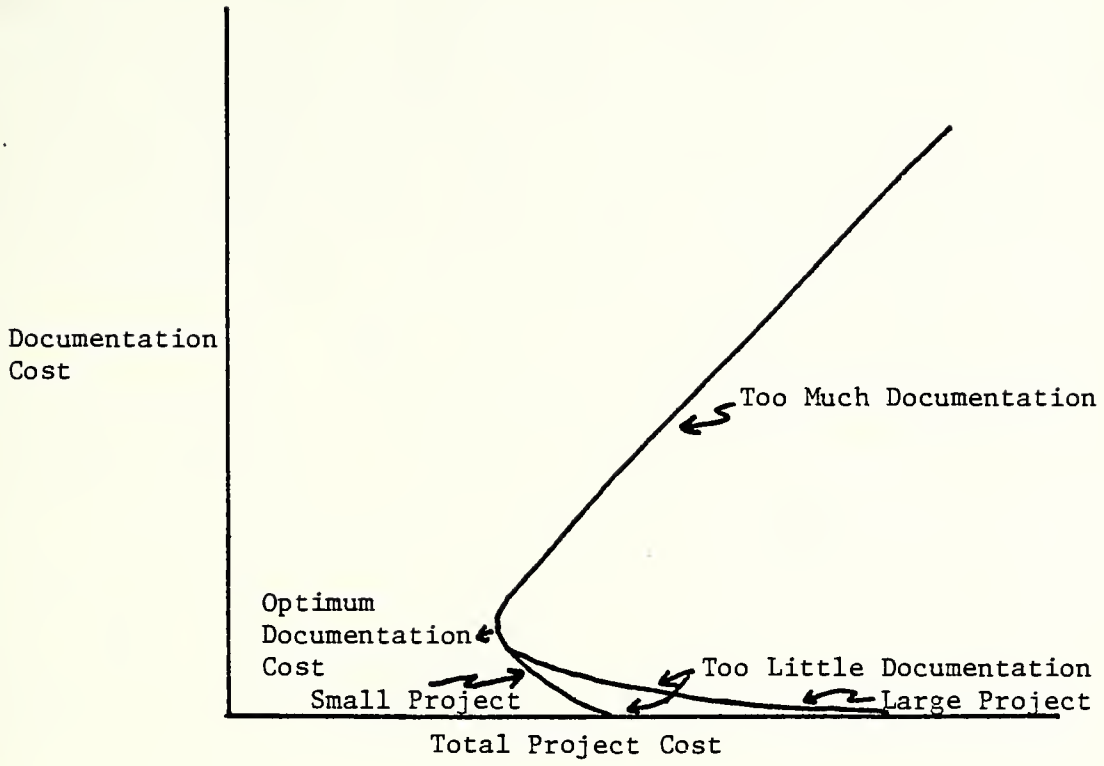


Figure 7. RELATION OF DOCUMENTATION COST TO TOTAL PROJECT COST

1. Engineering Estimation [57] (Also Bottom-up [11], Quantitative [56])

Engineering estimation is a generic term encompassing any methodology that systematically considers and evaluates all known, pertinent factors bearing on resource utilization. Variations of this method constitute the most highly used approach to software cost estimation. The basic procedure concerns breaking down a project effort into discrete work units (activities, tasks, etc.) and formulating separate estimates for each unit. Identification of an appropriate work breakdown structure is a critical step in this process. Costs in each separate activity can be aggregated into three cost centers--programmer productivity, computer time and elapsed project time. Once the difficulty of defining work units is resolved, the total number of work units is multiplied by a cost per unit factor or a productivity factor derived from estimates of software complexity and duration. Various software development factors unique to the project in question are often evaluated, reduced to a single weighting factor, and used to modify the derived estimate. The entire procedure is normally iterated several times during a project as more detailed data progressively becomes available. Engineering estimation is heavily reliant on the estimator's ability to evaluate each software development project in its unique internal development environment.

"A basic disadvantage of the many versions of this technique is the subjective assessment of the weighting factor used to modify the derived estimate. Also problematic is the previously determined cost per unit factor because it is not always clear what that cost includes (i.e., direct labor, direct labor plus overhead) and the unit (i.e., machine instructions, source statements) is often incomparable between projects." [56]

2. Parametric Relationships [57] (also ratio estimating [11])

These relationships have concentrated on the program design, coding and program testing phases. The most comprehensive work done in this area was a System Development Corporation Study in the mid 1960's sponsored by the Air Force Systems Command. This effort culminated in a massive regression analysis involving over 90 factors thought to be useful in predicting resource utilization. [55, 62] Determining which relationships are key to an individual project is the major operational problem with this approach.

3. Analogous Estimates [56] (also similarities and differences [11])

An initial task breakdown is accomplished to a level compatible with similar items in prior systems. Analogies are then drawn to known historic costs with adjustments made to account for technical differences. This method is heavily dependent upon the existence of an accurate, updated data base and/or upon the cost estimator's ability to recall relevant material and make proper analogies and adjustments. The analogy technique has been criticized for both the lack of a valid data base of historical performance, cost and schedule

data, and for the non-linear relationship between system costs and system size which confuses analogous comparisons.

4. Top-Down Estimation [11]

Wolverton [11] describes this approach as follows:

> "The estimator relies on the total cost of the large portions of previous projects that have been completed to estimate the cost of all or large portions of the project to be estimated. History coupled with informed opinion (or intuition) is used to allocate costs between packages."

Like analogous estimates, top-down estimating has been criticized for its dependence on data bases and the subjective skills of the estimator. [56]

5. Rules of Thumb

Many developed cost models have been reduced to rules of thumb for quick evaluations and checks against other estimates. Such rules can be quite useful if they are not relied upon solely. Table VI [57] summarizes a number of these rules.

6. The Putnam Model

a. Summary of Approach

An interesting approach to the software sizing and estimation problem was developed by Putnam [16] in his work with budgetary data from the U. S. Army Computer System Command. His effort is an extension of research by Norden [18] who found that man-loading for research and development projects can be linked to a project profile. Figure 8 [70] depicts individual manning phases tied to cycles underlying a summing "Project Profile" curve. Putnam represents Norden's

TABLE VI RULES OF THUMB (56)

DEVELOPMENT VARIABLE IMPACTING COSTS	SOURCE	RULE OF THUMB
.Allocation of Time to full Scale Development Activities	Aron(63)	Planning - 30% Coding - 20% Testing - 50%
	Brooks(15)	Planning - 33% Coding - 17% Testing - 50%
	Nelson(62)	Planning - 34.5% Coding - 18% Testing - 47.5%
	Wolverton (11)	Analysis/Design - 40% Code/Debug - 13% Test - 40%
	Marin (64)	Design Total System - 1 to 3 Man-Months (dependent on conditions and delays experienced) Design Computer Program System - 10% of Total Man-Months Design Programs - 1 Man-Month per 1000-2000 Machine Instructions Code Programs- 1 Man-Month per 5000 Machine Instructions Test Individual Programs - 20% of Test Effort Test Subsystem-Approximately 50% of Total Test Effort
.Computer Resource Requirements in Elapsed Time for FSD Activities	Aron & Arthur (65)	Six Hours/Programmer/Month (For Programmers with a Good Understanding of the Date Processing application) Eight Hours/Programmer/Month (for unfamiliar applications) Twelve Hours/Programmer/Month (For Real Time Projects)

DEVELOPMENT VARIABLE IMPACTING COSTS	SOURCE	RULE OF THUMB
	Meyers (66)	Three Hours/Month & Number Development Programmers through System Design Phase Twelve Hours/Month & Number Development Programmers through Implementation Phase Fifteen-twenty Hours/Month & Number Development Programmers through Integration & Test Phases
	Graver, et al(67)	Four Hours/Man-Month Twenty Hours/1000 Instructions
.Complexity of Software	Doty (68)	Real Time Applications are more complex Than non-real Time applications Scientific Applications are more complex Than Business Applications Operating Systems are more complex than Compilers Which are more complex than Applications software. Support Software is more complex than Applications Software.
	Aron(63)	Easy- Very few interactions with other System Elements Medium - Some interactions with Other System Elements Hard - Many interactions with Other System Elements

RULES OF THUMB (cont'd)

DEVELOPMENT VARIABLE IMPACTING COSTS	SOURCE	RULE OF THUMB
	Doty(68)	Thirty Pages per 1000 Lines of Source Code Five Pages per Man-Month
.Documentation	Marin(64)	Ten to Thirty-five Pages of Documentation/ 100 Lines Program Code Two Man-Months/User's Document Three to Five Pages/Man-Day for Draft, Using 750 -1250 Words/Page Twenty Pages/Man-Day for Technical Review Fifty Pages/Man-Day for Editing Ten to Fifteen Pages/Man-Day for Typing
	Geron(69)	Ten Percent (approximate) of Total Development Cost Thirty-Five to One Hundred Fifty Dollars/Non Automated Page

RULES OF THUMB (cont'd)

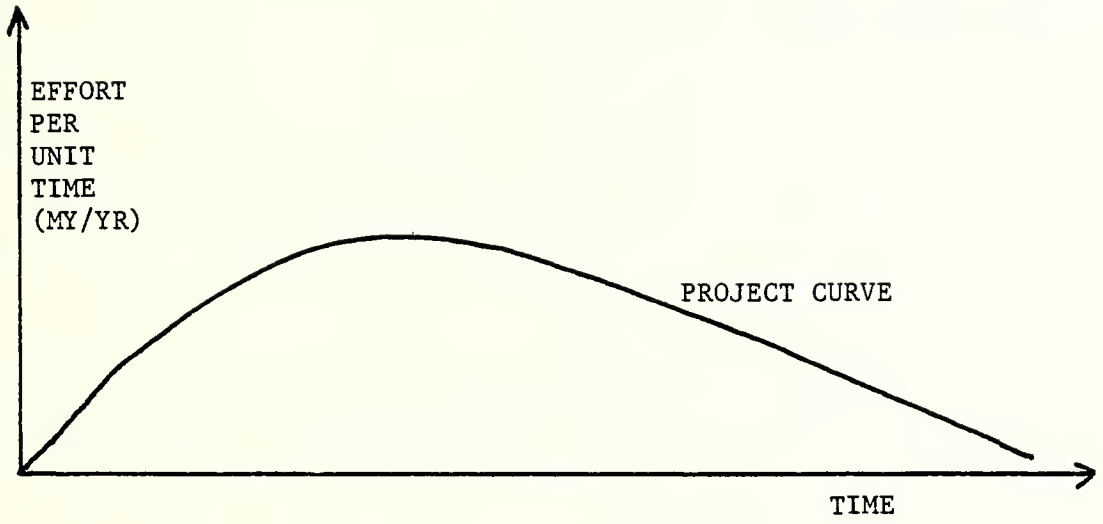


Figure 8. PROJECT PROFILE CURVE

model with the Rayleigh manpower equation which has been empirically determined to fit the project curve. The two important forms include both

- the derivative:

$$Y' = 2 Kate^{-at^2}$$

where Y' = man years of effort per year

K = total man years expended to develop system

a = "problem solving rate" parameter which determines curve shape

t = elapsed time in years

- and the integral:

$$K = K(1 - e^{-at^2})$$

where Y = cumulative man-years over time t .

Figure 9 [70] shows this Putnam-Rayleigh Model in both useful curve forms. Putnam further identifies the value

$$K / t_d^2$$

where t_d is the time to reach peak effort, as an indicator of the difficulty of a system in terms of the programming effort to produce it. To complete the cost prediction process, estimates of the two parameters of Putnam's model, K (the total life cycle man-years), and t_d (the time for the derivative curve to reach a maximum), are used to derive an equation giving the ordinates of the manpower requirement curve for a specific project. Yearly cost figures are then computed for the project by multiplying the ordinates of the

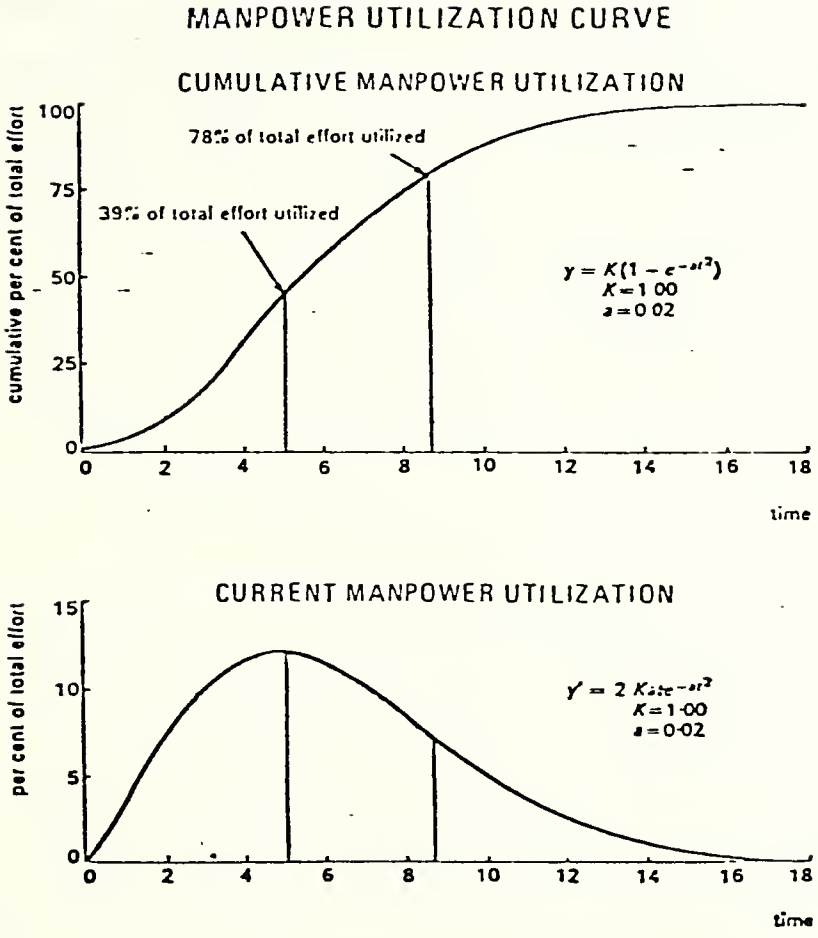


Figure 9. Life Cycle Integral and Derivative Curve Forms

manpower curve at each year by the average cost/man-year to arrive at a cost/year. These rates are then summed to find the cumulative cost. [56]

b. Management Implications According to Putnam [70]

(1) Life Cycle Size (K), Development Time (t_d) and difficulty (K/t_d^2) are natural parameters of a system. Each system is inherently stable and will be driven toward these parameters which constitute the minimum cost solution to the software design problem.

(2) Management cannot cut the development time of a project without increasing difficulty. All changes are biased to the negative direction. Development time cannot be arbitrarily set.

(3) If K, t_d and K/t_d^2 are accurately determined, a system can be designed-to-cost with little uncertainty.

c. Evaluation of the Putnam Model

Putnam has pointed out an impressive number of past projects conforming to his calculations. [16] If K and t_d can be confidently derived, the effort required to complete the estimation is minimal since the process can be easily automated. The breakdown of costs by time is an especially significant management aid.

General criticisms of the Putnam model include the following:

- Total reliance on man-years as a measure of work, thereby ignoring type of work. [57]

- Estimates of non manpower costs (e.g., computer time and overhead) inadequately addressed. [57]
- Accurate determination of K and t_d from historic data can be time consuming if such data is not easily available or in a usable form. [57]
- No conclusive data has been published concerning projects utilizing the Putnam Model as a major planning tool.
- No economic theory has been adequately presented to support Rayleigh curve fit for cost curves.
- Ease of automation may seduce weak managers to use inappropriately.

D. APPLYING THE CYCLOMATIC NUMBER

1. Utility

Many of the issues covered previously regarding software development and resource estimation suggest the importance of ordered program structures to both software quality and costs. If a method of measuring and controlling complexity in program structures is available, management will be able to accomplish the following [1]:

- Avoid error prone structures.
- Cut costs involved in extensive test and debug.
- Decrease time (and related costs) associated with extended development cycles.
- Assist in developing more standard modules.

- Facilitate resource estimation by increasing standardization and decreasing variance in programming productivity.
- More efficiently allocate resources by fitting manpower and schedule planning to complexity patterns.

The original work by McCabe [50] and supporting work by Schneidewind [1] indicate that the cyclomatic number offers a tool to effectively limit complexity. Its advantages include the following:

- Easy to understand and calculate.
- Requires information that can be developed during estimation process.
- Provides a finite number which can be used for planning.
- Facilitates formulation of appropriate test strategy, test input data and allocation of testing resource by
    - identifying independent substructures and
    - identifying heavily used logic paths.

2. Setting a Design Threshold

The particular upper bound to be set for the cyclomatic number is somewhat arbitrary and can probably be varied slightly from project to project. McCabe [35] suggests 10 as a reasonable upper limit. Since he found a variance among programmers from the 3 to 7 range to the 40 to 50 range, the imposition of such a limit would obviously radically alter

the approach of many programmers and would necessitate an introductory training period. The important point in implementing a cyclomatic number constraint is the ability of management to articulate the policy fully to programmers and enforce it by insisting that structures in violation be either modularized or redone.

3. Test Strategy and Resource Allocation

Even with an effective threshold, structures will naturally vary in complexity. In formulating the test procedure, more personnel, computer time and schedule time can be assigned to the structures with higher cyclomatic numbers in order to better allocate resources. Additionally, the directed graph analysis highlights portions of the software that are most heavily utilized in the logic flow and where program errors would be most damaging. Test input data can be selected to concentrate on these structures within the time constraints of the testing phase.

VI. SUMMARY AND CONCLUSIONS

- For various reasons relating to its nature and historic evolution, the process of large scale software development has been plagued by an inability of management to assess and control software complexity. Improving this ability will be a key factor in future project cost estimates, resource allocations, cumulative costs and software quality.

- In light of historic trends, the greatest potential for resource savings exists in the analysis/design and the test/integration phases of software development. Certain automated management tools have shown promise in these areas, but more experiential data is needed.

- As recognition of the importance of complexity has grown, a number of theorists and researchers have proposed methods of describing, estimating and measuring the extent of complexity's influence in individual programs.

- Perhaps because of the multifaceted nature of complexity, none of the proposed approaches has been shown to be sufficient in all cases. This fact may indicate the need for a "complexity profile," i.e., a comprehensive evaluation using more than one metric.

- An argument has been presented in support of the cyclomatic number (from McCabe's Directed Graph application to modeling software) as a useful tool for control of complexity and the allocation of resources.

LIST OF REFERENCES

1. Schneidewind, N. F., and Hoffman, H.-M. "An Experiment in Software Error Data Collection and Analysis," IEEE Transactions on Software Engineering, May 1979.

2. Naval Air Development Center Report NPS555a75021, Structure and Error Detection in Computer Software, by G. H. G. H. Branley, et. al., February 1975.

3. SREM Operations Manual, TRW, 1976.

4. Goldberg, J., Cheatham, T. E., et. al., Proceedings of a Symposium on the High Cost of Software, Stanford Research Institute, 1973.

5. Boehm, B. W., "Software Engineering," IEEE Transactions on Computers, December 1976.

6. Cooper, J. D. (ed.), Software Quality Management, Petrocelli, 1979.

7. Manley, J. H., "Software Life Cycle Management: Dynamics Theory," 2nd Software Life Cycle Management Workshop, August 1978.

8. Naval Weapons Center Report NPS52-78-001, A Study of Alternatives for VSTOL Computer Systems, by V. R. Kodres, et. al., April 1978.

9. Daly, E. B., "Management of Software Development," IEEE Transactions on Software Engineering, May 1977.

10. McHenry, R. C. and Watson, C. E., "Software Life Cycle Management: Weapons Process Developer," IEEE Transactions on Software Engineering, July 1978.

11. Wolverton, R. W., "Developing Large Scale Software," IEEE Transactions on Software Engineering, June 1974.

12. Weinberg, G. W., The Psychology of Computer Programming, Van Nostrand Reinhold Co., 1972.

13. Paretta, R. L. and Clark, S. A., "Management of Software Development," Journal of Systems Management, April 1974.

14. Ridge, W. J. and Johnson, L. E., Effective Management of Computer Software, Dow Jones-Irwin, 1973.

15. Brooks, F. P., The Mythical Man-Month: Essays of Software Engineering, Adison Wesley, 1974.

16. Putnam, L. H., "A Macro-Estimating Methodology for Software Development," Thirteenth IEEE Computer Society International Conference, September, 1976.

17. Snyder, T. R., "Rate Charting," Datamation, November 1976.

18. Norden, P. V., "Useful Tools for Project Management," Management of Production, April 1977.

19. Thibodeau, R. and Dodson, E. N., "The Implications of Life Cycle Phases Interrelationships for Software Cost Estimating," Second Software Life Cycle Management Workshop, August 1978.

20. Hibbard, P. G. and Schuman, S. A., Constructing Quality Software, North Holland Publishing, 1978.

21. Cave, W. C. and Salisbury, A. B., "Controlling the Software Life Cycle - The Project Management Task," IEEE Transactions on Software Engineering, July 1978.

22. Belford, P. C., et. al., "Specifications - a Key to Effective Software Development," Second International Conference on Software Engineering, October 1976.

23. Yeh, R. T. (ed.), Current Trends in Programming Methodology, Vol. I, Prentice-Hall, 1977.

24. Ballistic Missile Defense Advanced Technology Center Report DASG60-75-C-0022, Software Requirements Methodology Final Report - Vol. I, by TRW, August 1977.

25. Teichroew, D. and Hershew, E. A., "A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems," IEEE Transactions of Software Engineering, January 1977.

26. Ross, D. T., "Structured Analysis: A Language for Communicatig Ideas, IEEE Transactions on Software Engineering, January 1977.

27. Willis, J., "DAS: An Automated System to Support Design Analysis," paper presented at 12th Annual Conference on Circuits, Systems & Computers, 1979.

28. Naval Air Development Center Report SDGTM 73-CVTSC-041, "Use of the Top-Down Approach and Structured Programming on CVTSC Software," May 1973.

29. Comer, D. and Halstead, M. H., "A Simple Experiment in Top-Down Design," IEEE Transactions on Software Engineering, March 1979.

30. Carrow, J. C., "Structured Programming: From Theory to Practice," Second Software Life Cycle Management Workshop, August 1968.

31. Nassi, I. and Schneiderman, B., "Flowchart Techniques for Structured Programs," SIGPLAN Notices, August 1973.

32. Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R., Structured Programming, Academic Press, 1972.

33. Baker, F. T. and Mills, H. D., "Chief Programmer Teams," Datamation, December 1973.

34. Baker, F. T., "Chief Programmer Team Management of Production Programming," IBM Systems Journal, No. 1, Vol. 11, 1972.

35. Naval Air Systems Command Technical Note NADC-76044-50, Improved Software Productivity for Military Computer Systems through Structured Programming, by R. J. Pariseau, 1978.

36. Ramamoorthy, C. V. and Ho, S. F., "Testing Large Software with Automated Software Evaluation Systems," IEEE Transactions on Software Engineering, March 1975.

37. Hoffman, H. -M., An Experiment in Software Error Occurrence and Detection, Masters Thesis, Naval Postgraduate School, June 1977.

38. Yin, B., "Investigation of Software Design Structures Which Enhance Testability," Book 1, Section 6, Hughes Aircraft Co., June 1979.

39. Freemen, P. and Irvine, C. A., "Requirements Analysis and Definition for Software Development," lecture notes, 1978.

40. Cooper, J. D., "Corporate Level Software Management," IEEE Transactions on Software Engineering, July 1978.

41. Szygenda, S. A. and Thompson, E. W., "Modeling and Digital Simulation for Design Verification and Diagnosis," IEEE Transactions on Computers, December 1976.

42. Curtis, B., et. al., "Some Distinctions Between the Psychological and Computational Complexity of Software," Second Software Life Cycle Management Workshop, Computer Society, August 1978.

43. Jones, N. P., Computability Theory, Academic Press, 1973.

44. Miller, R. E. and Thatcher, J. W., Complexity of Computer Computations, Plenum Press, 1972.

45. Goodman, L. I., Complexity Measures for Programming Languages, M. S. Thesis, Department of Electrical Engineering, MIT, September 1971.

46. Savage, J. E., The Complexity of Computing, Wiley & Sons, 1976.

47. Gilb, T., Software Metrics, Winthrop, 1977.

48. Quantitative Software Models, SRR-1, Prepared by IIT Research Institute for Rome Air Development Center, March, 1979.

49. Thayer, T. A., et. al., "Software Reliability Study," Rome Air Development Center Report Number RADC-TR-76-L38, August 1976.

50. McCabe, T. J., "A Complexity Measure," IEEE Transactions on Software Engineering, IEEE Computer Society, December 1976.

51. McCabe, T. J., "Software Complexity Measurement," Second Software Life Cycle Management Workshop, IEEE Computer Society, August 1978.

52. Halstead, M. H., Elements of Software Science, Elsevier, 1977.

53. Halstead, M. H., "Guest Editorial on Software Science," IEEE Transactions on Software Engineering, IEEE Computer Society, March 1979.

54. Curtis, B., et. al., "Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics," IEEE Transactions on Software Engineering, IEEE Computer Society, March 1979.

55. Wolverton, R. W., Personal interview at TRW, Redondo Beach, California, August 1979.

56. Finfer, M. F. and Mish, R. K., Software Acquisition Management Guidebook: Software Cost Estimating and Measurement, Report number ESD-TR-78-140, prepared by System Development Corporation.

57. Andies, A. W., Estimating Computer Software Development Costs, Study Report 75.1, Defense System Management College, June 1978.

58. Boehm, B.W., "Software and Its Impact: A Quantitative Assessment," Datamation, May 1973.

59. Lehman, M. M., "Laws and Conservation in Large-Program Evolution," Second Software Life Cycle Management Workshop, IEEE Computer Society, August 1978.

60. Wood, D. L., "Data Processing Value Engineering," Society of American Value Engineering Proceedings, May 1973.

61. Clapp, J. A., A Review of Software Cost Estimation Methods, Report ESD-TR-76-371, prepared by Mitre Corporation.

62. Nelson, E. A., Managerial Handbook for the Estimation of Computer Programming, SDC Report TM-3225/000/01, March 1967.

63. Aron, J. D., "Estimating Resources for Large Programming Systems," IBM Federal Systems Center, 1969.

64. Marin, L., Estimation of Resources for Computer Programming Projects, Masters Thesis, University of North Carolina at Chapel Hill, 1973.

65. Aron, J. D. and Arthur, R. W., "Computer Resource Requirements for Programming Development," IBM, 1975.

66. Meyers, G. J., "Estimating the Costs of Programming Systems," IBM Technical Report TR 00.2316, IBM, May 1972.

67. Graver, C. A., "Cost Reporting Elements and Activity Cost Tradeoffs for Defense System Software, Volume I: Study Results," Report CR-1-721, General Research Corp., 1977.

68. "Software Cost Estimation Study, Volume II: Guidelines for Improved Software Cost Estimation," Report RADC TR 77-220, Doty Associates, Inc., February 1977.

69. Geran, D. B., "Summary Notes of a Government/Industry Software Sizing and Costing Workshop," Bedford, Mass., October 1973.

70. Putnam, L. H. and Wolverton, R. W., Quantitative Management: Software Cost Estimating Tutorial, IEEE Computer Society, 1977.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	2
2.	Library, Code 0142 Naval Postgraduate School Monterey, California 93940	2
3.	Defense Logistics Studies Information Exchange U. S. Army Logistics Management Center Fort Lee, Virginia 23801	1
4.	Professor C. R. Jones, Code 54Js Chairman, Department of Administrative Sciences Naval Postgraduate School Monterey, California 93940	1
5.	Professor N. F. Schneidewind, Code 54 Ss Department of Administrative Sciences Naval Postgraduate School Monterey, California 93940	1
6.	LTC R. S. Miller, USA, Code 55Mu Department of Operations Research Naval Postgraduate School Monterey, California 93940	1
7.	LCDR J. N. Harris, USN 3112 Sloat Road Pebble Beach, California 93953	2

Thesis
H2895 Harris
c.1 Complexity as a factor of quality and cost in large scale software development.

185577